Poul Klausen

JAVA 13

Distributed programming and Java EE

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 13: DISTRIBUTED PROGRAMMING AND JAVA EE

SOFTWARE DEVELOPMENT

Java 13: Distributed programming and Java EE: Software Development
1st edition

ISBN 978-87-403-2098-5
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the thirteenth in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. This book is a continuation of the subjects in Java 11, but with focus on the development of enterprise applications. They are programs that performs their work using components or services hosted on different computers and located on different places, which communicate and coordinate their work over a network. Primary topics are Java EE technologies as JPA, EJB and Web Services. The book requires knowledge of programming of web applications similar to what has been dealt with in the books Java 11 and Java 12.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

1. NetBeans as IDE for application development
2. MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
3. GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not "how to write" or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

In the previous two books, I have been working on developing web applications, and this book is a natural continuation, but focusing on the development of distributed programs. They are programs that perform their work using components or services hosted on different computers, which communicate and coordinate their work over a network. The whole idea is that these components largely have their own lives and live independently of each other and generally without knowing the clients who uses the services they provide. In fact, it is not easy to precisely define what we understand with a distributed program, but typically we will understand a program that is characterized by

1. that there are several independent computing units (computers), each of which has their own local memory
2. the computing units communicate by sending messages to each other over a network

The goal of a distributed program (or system) may be to solve a particular task, and users will then perceive the family of computing units as a single device (or system), but instead, they may be a service-oriented system that offers a range of services which users can use and perceive as shared resources. Distributed programs will also be characterized by parallelity, where multiple activities are performed simultaneously on multiple computers and locations.

You can also sometimes see distributed programming described as a further development of object-oriented programming. An object-oriented program consists of objects with each of their internal data and logic hidden from the outside world and only known through the services that the object makes available in the form of public methods. These objects then work together to solve the task to be solved. Similarly, you can see a distributed program as a program that solves the task by providing a number of services that live independently of each other on computers around the world where applications can use them if they otherwise will be allowed.

In addition to showing how to develop distributed programs in practice (or perhaps more in relation to this), the goal is also to describe more specific technologies as in headings are

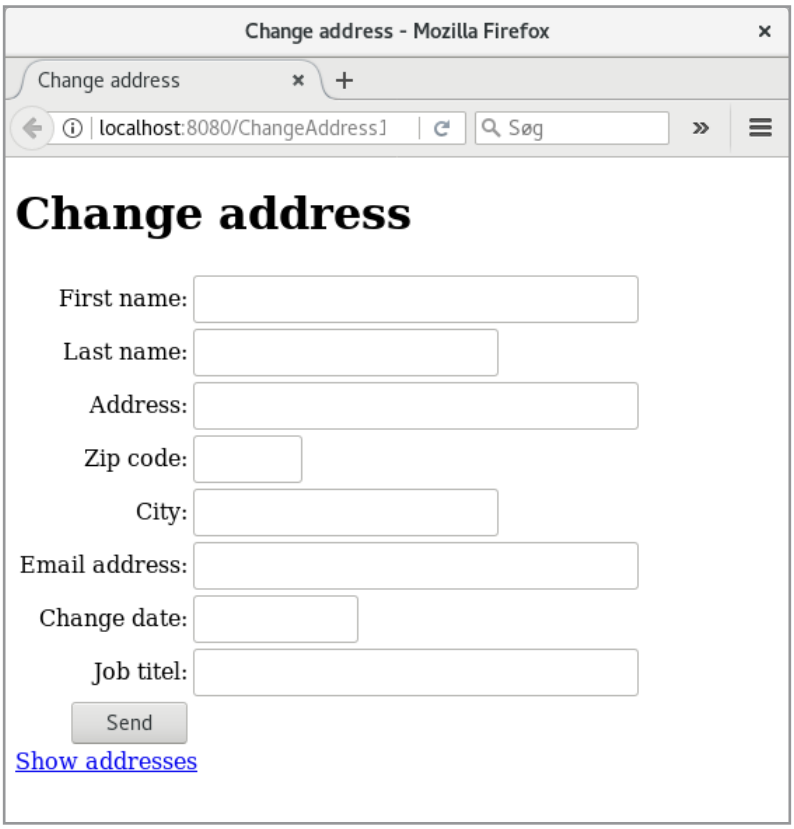
- Java Persistence API
- Java Enterprise Beans
- Contexts and Dependency Injection
- Java Web Services
- RESTFul web services
- Java EE security

As it appears, the book describes a number of Java APIs, and much of the substance is directly related to NetBeans and GlassFish.

2 JAVA PERSISTENCE API

In the previous books (from Java 6 onwards) and including Java 11 on web applications, I have considered several examples that uses databases. It has always included programming an object oriented encapsulation of the database in the form of model classes for each of the database tables and one or more classes that encapsulates the SQL statements that will manipulate the content of the relational database. This has meant that for each example, many lines have been required to write the Java code, and although it may be affordable, as it is in principle the same thing that should happen every time, it can be done easier (and better) , as Java EE offers an API called *Java Persistence API*, which has the exact task of offering an object-oriented interface to a relational database. The aim of this chapter is to give an introduction to this API, and the content largely characterizes "how to do".

I will start with a copy of the project *ChangeAddress3* from the book Java 11, where I have called the copy *ChangeAddress1*. If you open the application in the browser, you get the following window, where you can to enter an address, and if you click on the bottom link, you get another window that shows an overview of the addresses that were entered.

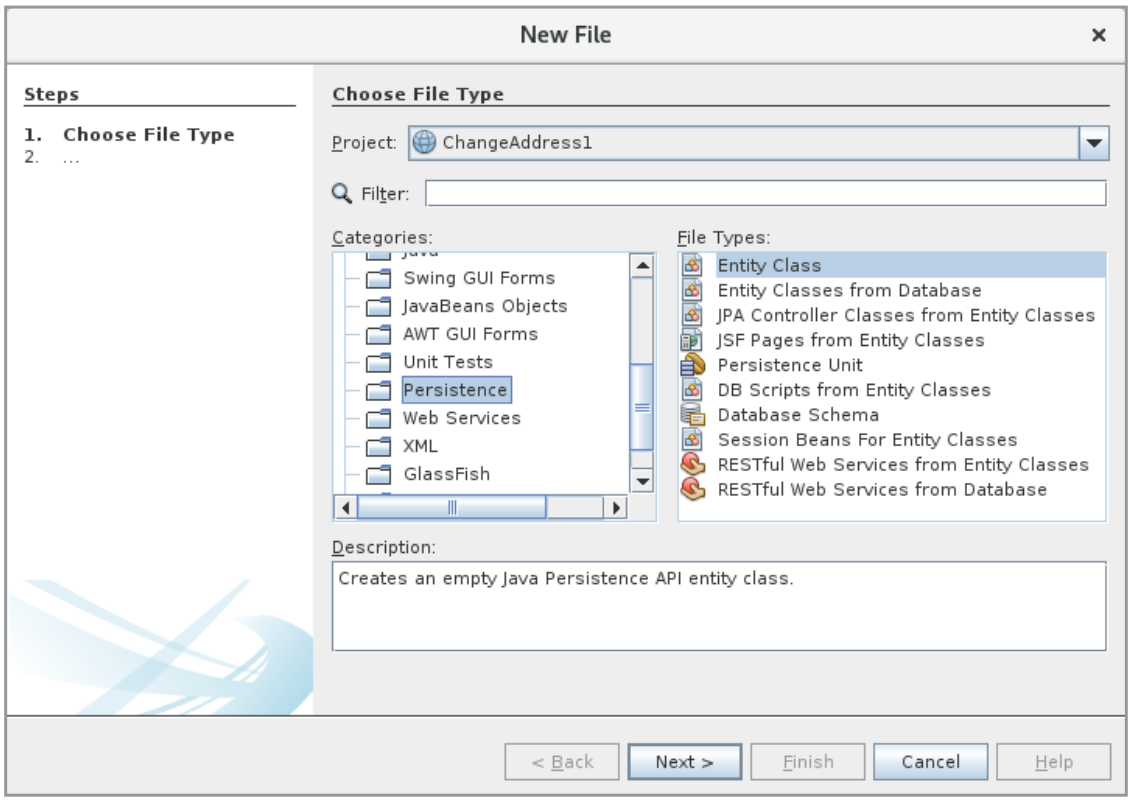


When entering addresses, they are stored in a list of *Person* objects. However, data is not stored persistently. As I have shown in previous examples, it is relatively simple to create

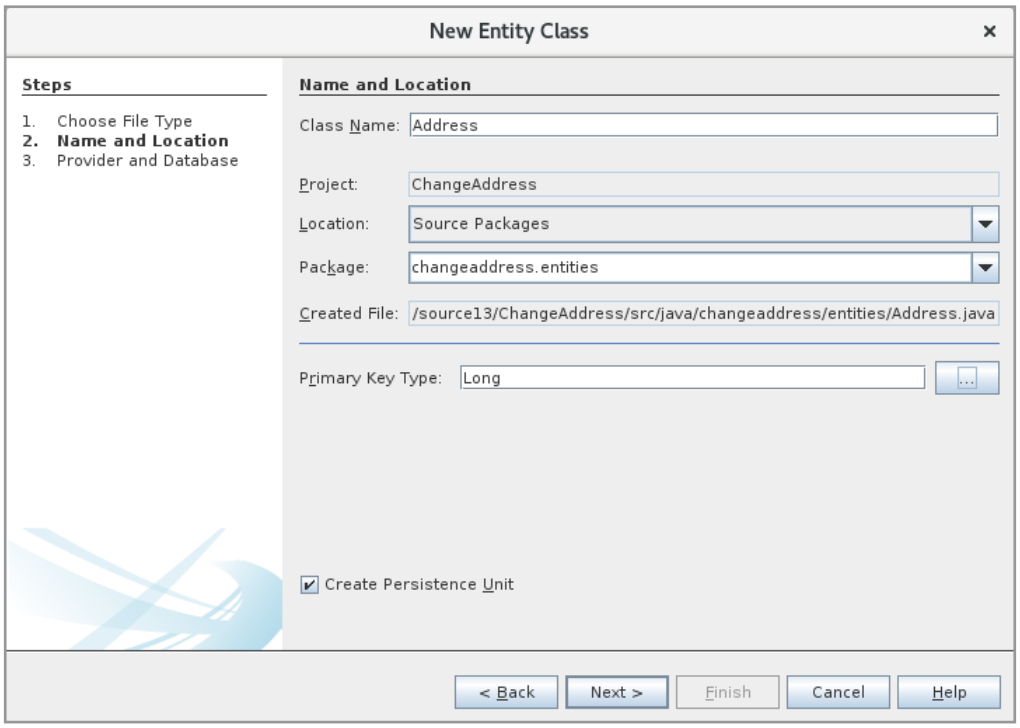
a database with a table, and then save data in this table. In this example, I will show you how to do it using JPA.

In this example, I want to use *JavaDB* as database product. Primarily to show this database, since it is useful and easy to use in many contexts where you need a simple database. It is a standard SQL database server, supported by the Glassfish server, which does not require any special installation, and the product is also supported by NetBeans.

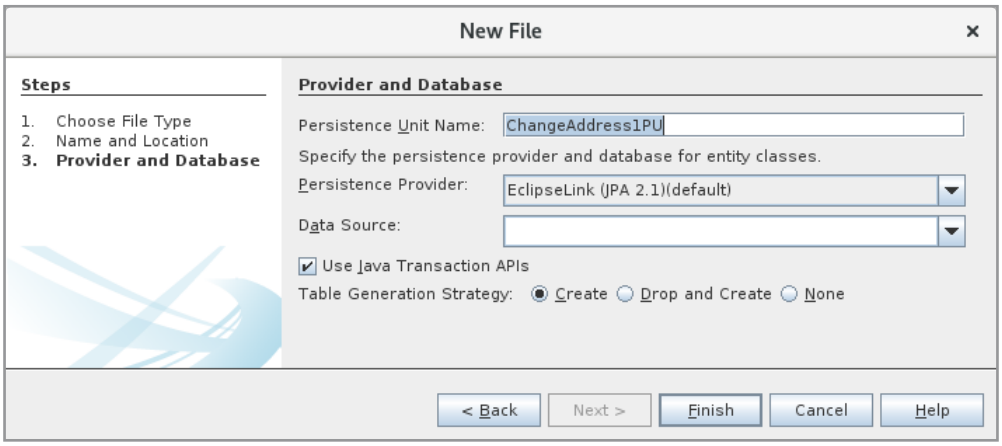
The method is basically defining an entity class for each database table, and I would like to start by associating a class to the *model* package. It must be an *Entity Class*:



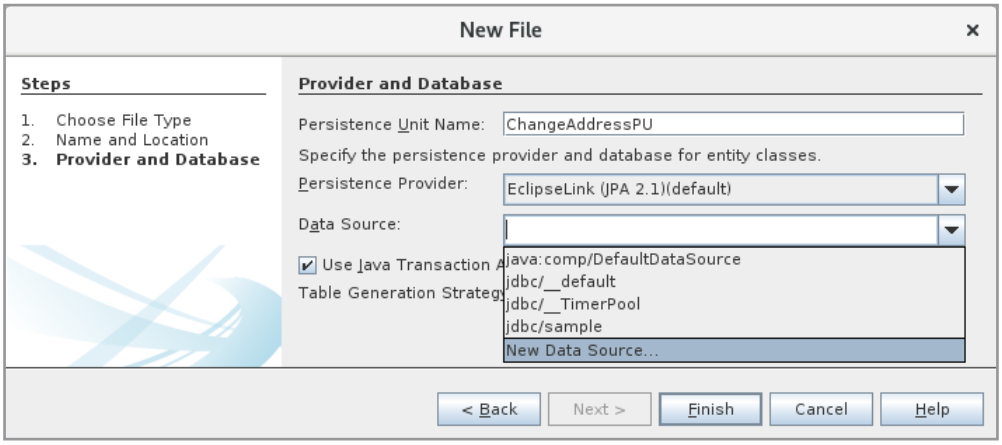
and it should be called *Address*:



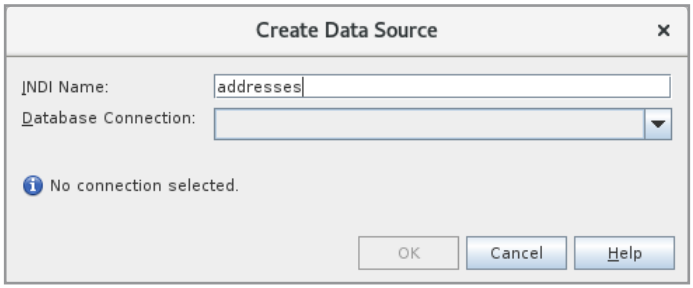
You should note that the correct package name has been selected (can be entered if it is not in the list) and that the *Create Persistence Unit* has been ticked. A persistence unit is an XML configuration file called *persistence.xml*, created by NetBeans. When you click *Next* in the window above, you get a window to initialize this XML file. You can change the name if you wish, and note that *Use Java Transaction APIs* have been ticked. By default, *Create* has been selected, which means that the database must be created if it is not already.



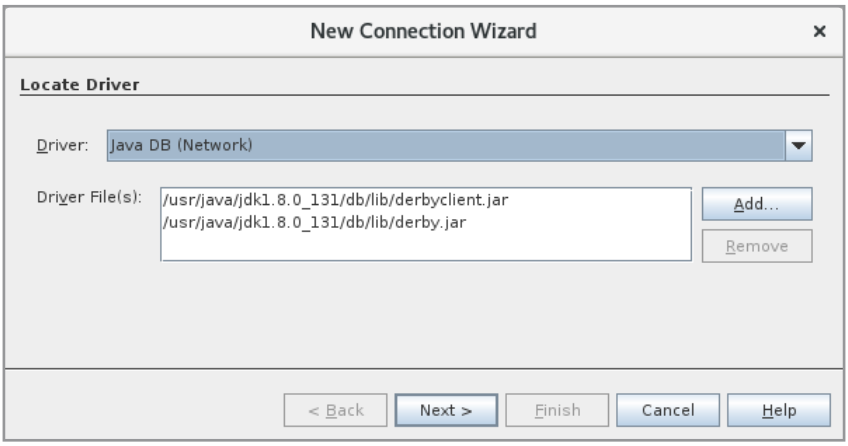
However, the most important is the *Data Source* field, which is a reference to the current database. Here you can choose an existing data source, or you can create a new one:



If you click on *New Data Source*, you get a window to create a data source:



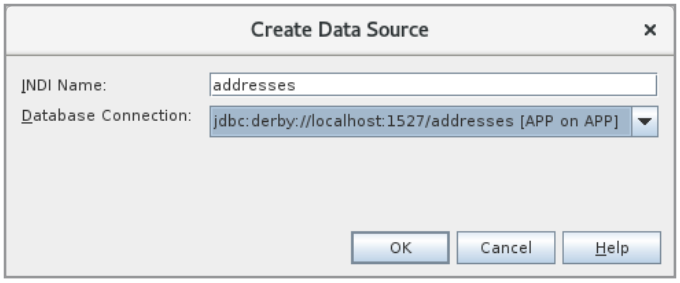
where I have entered the name *addresses*. A connection to the database must also be defined. Here you can either choose an existing one or create a new one. In this case, you must create a new one, and choose a driver where I have selected *Java DB (Network)*:



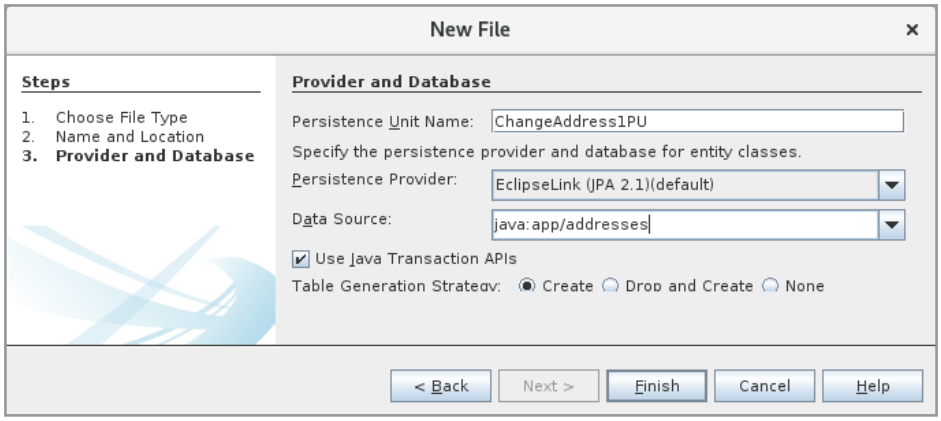
When clicking Next, enter information about the database:

The database server – the Glassfish server – runs on localhost and uses 1527 as port number. I have chosen that the database should be called *addresses*. Java DB has a default user called

APP, and I have entered a password (1234). Then I click *Next* twice and finally *Finish*, after which the data source has been created:



When I click *OK*, the Entity class is created:



and then I click *Finish*, NetBeans has created the following class (where I have removed comments and modified *equals()* a bit):

```
package changeaddress.models;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Address implements Serializable
{
 private static final long serialVersionUID = 1L;
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private Long id;

 public Long getId()
 {
  return id;
 }
```

```
 public void setId(Long id)
 {
  this.id = id;
 }

 @Override
 public int hashCode()
 {
  int hash = 0;
  hash += (id != null ? id.hashCode() : 0);
  return hash;
 }

 @Override
 public boolean equals(Object object)
 {
  if (!(object instanceof Address)) return false;
  return id.equals(((Address)object).id);
 }

 @Override
 public String toString()
 {
  return "changeaddress.models.Address[ id=" + id + " ]";
 }
}
```

The result is a class with a single property *ID* of the type *Long*. You should note that this property is decorated with two annotations, first indicating that it is a primary key, while the other tells that the value should be assigned by the database server as an auto number. Finally, you should note that the class is also decorated with an annotation that tells that it is an entity class. The class must then be expanded with the following properties:

```
private String firstname;
private String lastname;
private String address;
private String code;
private String city;
private String email;
private String date;
private String title;
```

with associated get and set methods. Then I have deleted the model class *Person*, and in *IndexController*, all references to *Person* must be changed to *Address*, and the code in the *add()* method should be deleted. After that, the program can be opened in the browser again.

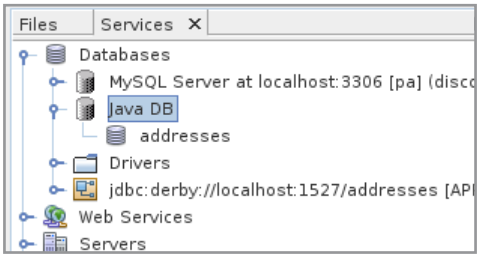
In *IndexController*, you must add some additional statements:

```
package changeaddress.beans;

import java.util.*;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
import javax.persistence.*;
import javax.transaction.*;
import javax.annotation.*;
import javax.persistence.criteria.CriteriaQuery;

import changeaddress.models.*;

@Named(value = "indexController")
@SessionScoped
public class IndexController implements Serializable
{
 @PersistenceUnit
 EntityManagerFactory emf;
 @PersistenceContext
 EntityManager em;
```

```
@Resource
UserTransaction utx;

private Address person = new Address();
private List<Address> persons = new ArrayList();

…

public List<Address> getPersons()
{
 try
 {
  CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
  cq.select(cq.from(Address.class));
  Query q = em.createQuery(cq);
  return persons = q.getResultList();
 }
 catch (Exception ex)
 {
 }
 return persons;
}

public void add()
{
 try
 {
  utx.begin();
  em.persist(person);
  utx.commit();
  person = new Address();
 }
 catch (Exception ex)
 {
  try
  {
   utx.rollback();
  }
  catch (Exception e)
  {
  }
 }
}
```

At the start of the class, three objects are defined each decorated with an annotation. The exact meaning of this notation is explained in chapter 4, but as example, the variable *em*

has the type *EntityManager* and the preceding annotation means that, on the basis of the *Persistence Unit* for the project, JPA instantiates an object to the type *EntityManager* that is a JPA type, which represents a database table and offers services that can perform database operations. The last variable *utx* defines a transaction for a database operation. These objects are used in the methods *getPersons()* and *add()*. For example, if you look at the last one, it starts a transaction and then the method *persist()* writes the object *person* to the database. Before the application can be used, the database must be created. It happens under *Services* by right-click *Java DB*:



and here you choose *Create Database*. With these changes, the program works the same as in the book Java 11, but now the addresses are stored in a database table.

2.1 AN IMPROVED ADDRESS PROGRAM

In this section, I want to show another version of the above program that has been changed in two areas:

1. The database is this time an existing MySQL database.
2. The database operations have been moved to a database controller.

Before I get the task done, I have to create the database:

```
use mysql;

create database addresses;

use addresses;

create table address (
 id int not null auto_increment,
 firstname varchar(50),
 lastname varchar(30),
 address varchar(50),
```

```
 code varchar(4),
 city varchar(30),
 mail varchar(50),
 date varchar(10),
 title varchar(50),
 primary key (id)
);
```

and the result is a database with the same content as in the previous example.

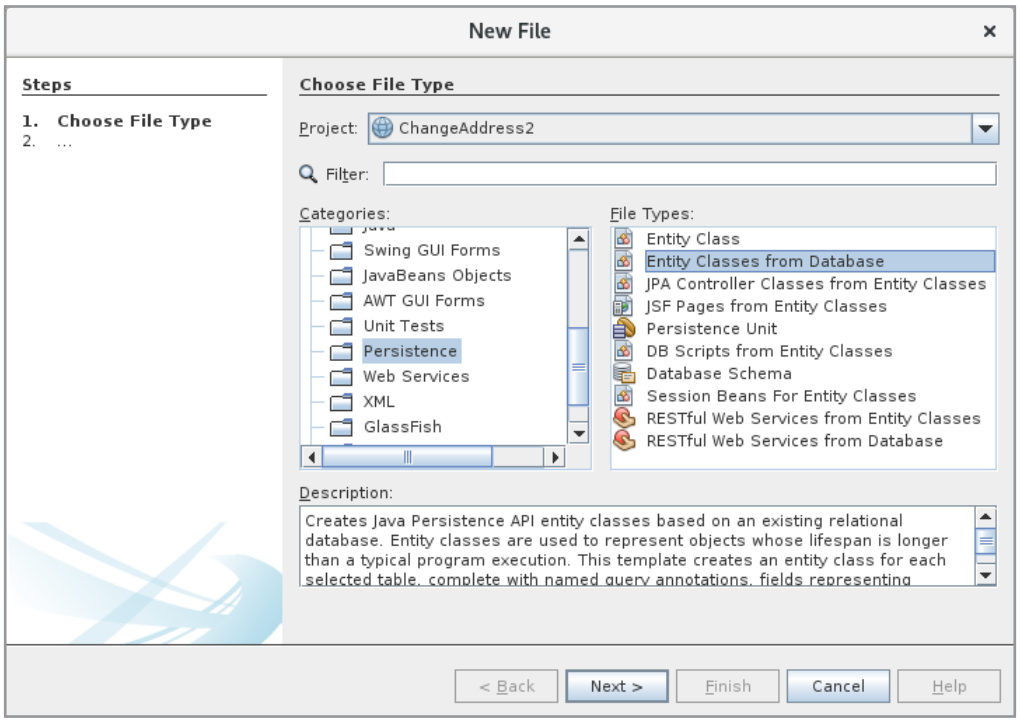
The starting point is a copy of the above program, called *ChangeAddress2*. First, I have deleted the following files:

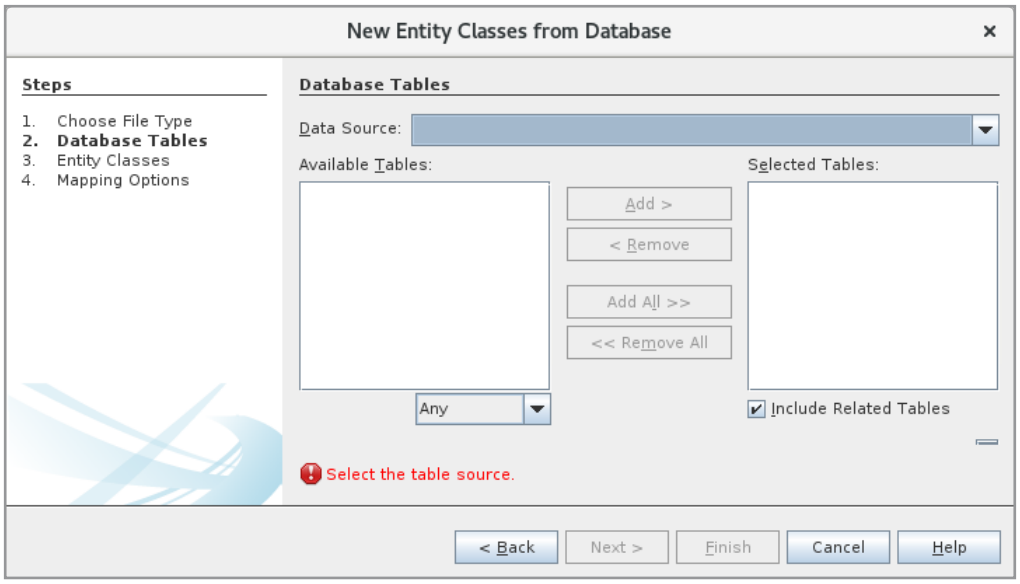
1. The model class *Address*
2. The configuration file *glassfish-resources.xml*
3. The configuration file *persistence.xml*

Then I have created the model class again (and with the same name), but this time as an *Entity Class from Database* (see below). When I click *Next*, I get the window below, where I have selected a *Data Source*. This happens in the same way as in the previous example, but

this time, for database connection, a *MySQL driver* must be selected to specify the database (*addresses*), database user and password.

Once done, the data source's (as I have called *addresses*) tables are shown under *Available Tables*, and it should then be added to *Selected Tables*. Then click *Next* twice and accept all settings and finally click *Finish*. Then, the class *Address* is created again.



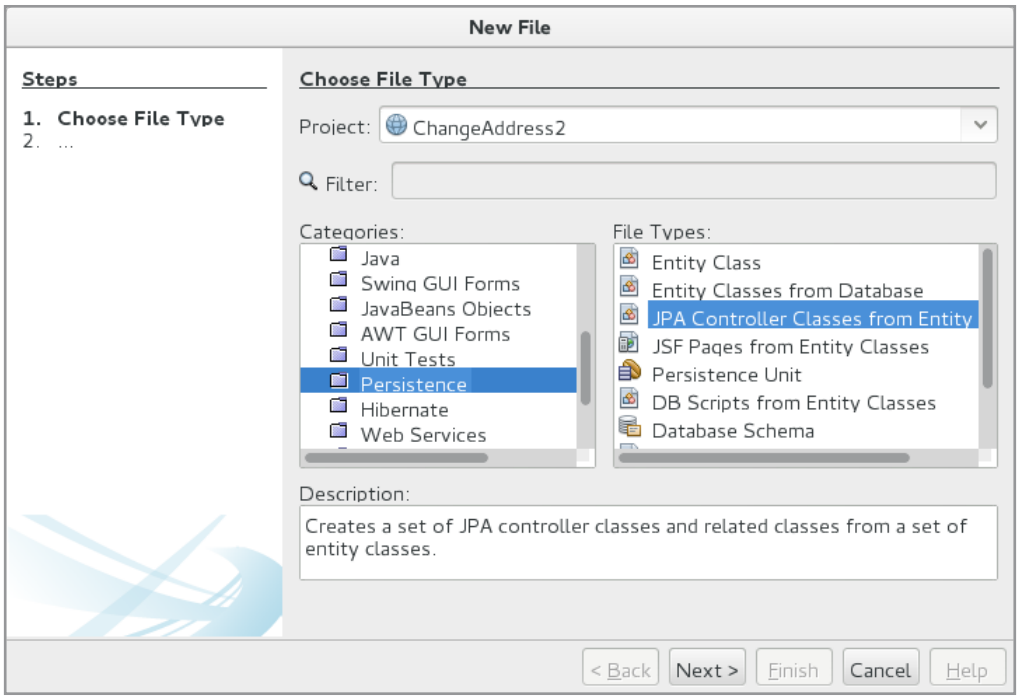


There is a small problem, however, because the class's *Address* property for email address is called *mail* instead of *email*, and therefore a few names must be changed in *IndexController*.

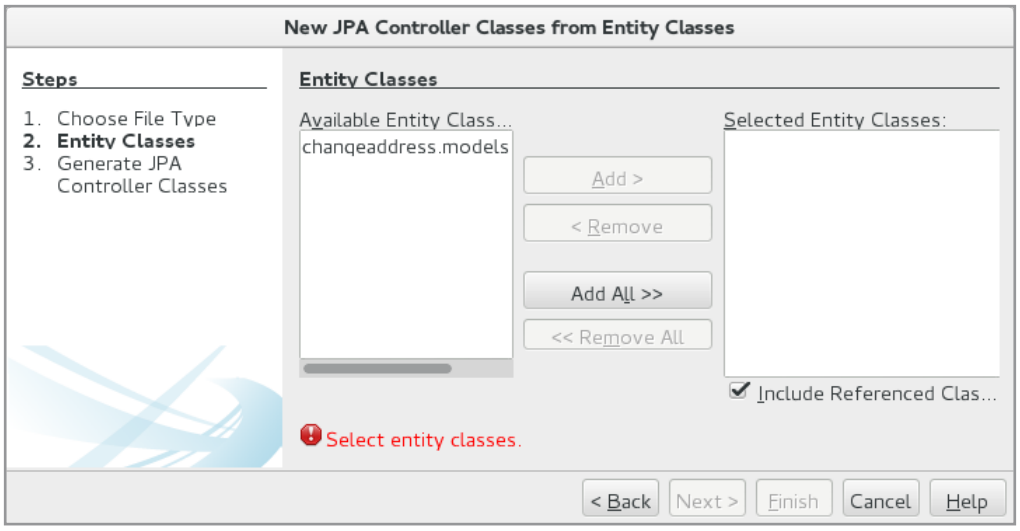
Then the program can run again and the only difference is that the application now uses the MySQL database.

You should examine the class *Address*, where many annotations have been added (all properties are decorated), but you can easily understand the meaning. You must note that all annotations are determined by NetBeans by retrieving information from the database. Also try to examine the glassfish resource *resources.xml* configuration file.

I will make a single change. With an *EntityManager* object and a *UserTransaction* object, you can perform database operations without the need to write very much and without knowing SQL. Since it's the same thing that's going to happen every time, you can encapsulate these operations into a class that can be used no matter what database table it is. This happens by adding a database controller (here to *changeaddress.models*), which is a class of the type *JPA Controller Classes from Entity*:



When you click Next, you must select the entity class for which the object should be controller for:



In this case, there is only one option, and when added to *Selected Entity Classes*, you can create the controller class. The class is called *AddressJpaController* and it encapsulates what is necessary to maintain the database table as well as methods so that you can read the row with a particular primary key as well as a few methods to retrieve multiple rows. The code is shown below:

```
package changeaddress.models;

import changeaddress.models.exceptions.*;
import java.io.Serializable;
import java.util.List;
import javax.persistence.*;
import javax.persistence.criteria.*;
import javax.transaction.UserTransaction;
public class AddressJpaController implements Serializable
{
 private UserTransaction utx = null;
 private EntityManagerFactory emf = null;

 public AddressJpaController(UserTransaction utx, EntityManagerFactory emf)
 {
  this.utx = utx;
  this.emf = emf;
 }

 public EntityManager getEntityManager()
 {
  return emf.createEntityManager();
 }

 public void create(Address address) throws RollbackFailureException, Exceptior
 {
  EntityManager em = null;
  try
  {
   utx.begin();
   em = getEntityManager();
   em.persist(address);
   utx.commit();
  }
  catch (Exception ex)
  {
   try
   {
    utx.rollback();
   }
   catch (Exception re)
   {
    throw new RollbackFailureException(" … ", re);
   }
   throw ex;
  }
  finally
  {
```

```
   if (em != null)
   {
    em.close();
   }
  }
 }

 public void edit(Address address) throws NonexistentEntityException,
  RollbackFailureException, Exception
 {
  EntityManager em = null;
  try
  {
   utx.begin();
   em = getEntityManager();
   address = em.merge(address);
   utx.commit();
  }
  catch (Exception ex)
  {
   try
   {
    utx.rollback();
   }
   catch (Exception re)
   {
    throw new RollbackFailureException(" … ", re);
   }
   String msg = ex.getLocalizedMessage();
   if (msg == null || msg.length() == 0)
   {
    Integer id = address.getId();
    if (findAddress(id) == null)
    {
     throw new NonexistentEntityException(" … ");
    }
   }
   throw ex;
  }
  finally
  {
   if (em != null)
   {
    em.close();
   }
  }
 }
```

```
public void destroy(Integer id) throws NonexistentEntityException,
 RollbackFailureException, Exception
{
 EntityManager em = null;
 try
 {
  utx.begin();
  em = getEntityManager();
  Address address;
  try
  {
   address = em.getReference(Address.class, id);
   address.getId();
  }
  catch (EntityNotFoundException enfe)
  {
   throw new NonexistentEntityException(" … ", enfe);
  }
  em.remove(address);
  utx.commit();
 }
 catch (Exception ex)
 {
  try
```

```
   {
    utx.rollback();
   }
   catch (Exception re)
   {
    throw new RollbackFailureException(" … ", re);
   }
   throw ex;
  }
  finally
  {
   if (em != null)
   {
    em.close();
   }
  }
}

public List<Address> findAddressEntities()
{
 return findAddressEntities(true, -1, -1);
}

public List<Address> findAddressEntities(int maxResults, int firstResult)
{
 return findAddressEntities(false, maxResults, firstResult);
}

private List<Address> findAddressEntities(boolean all, int maxResults,
 int firstResult)
{
 EntityManager em = getEntityManager();
 try
 {
  CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
  cq.select(cq.from(Address.class));
  Query q = em.createQuery(cq);
  if (!all)
  {
   q.setMaxResults(maxResults);
   q.setFirstResult(firstResult);
  }
  return q.getResultList();
 }
 finally
 {
  em.close();
 }
}
```

```
 public Address findAddress(Integer id)
 {
  EntityManager em = getEntityManager();
  try
  {
   return em.find(Address.class, id);
  }
  finally
  {
   em.close();
  }
 }

 public int getAddressCount()
 {
  EntityManager em = getEntityManager();
  try
  {
   CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
   Root<Address> rt = cq.from(Address.class);
   cq.select(em.getCriteriaBuilder().count(rt));
   Query q = em.createQuery(cq);
   return ((Long) q.getSingleResult()).intValue();
  }
  finally
  {
   em.close();
  }
 }
}
```

The code fills, but the class is easy enough to understand and the most important thing is that everything about updates to the database is encapsulated in methods, including the necessary logic for rollback. In addition, the class defines some simple *find()* methods that can return an object with a specific *id* or list of objects that typically will be all objects. These *find()* methods uses a private *find()* method, and here you should note the *CriteriaQuery* and *Query* types as explained below, but based on the names (and the application) you can see that these are the types that will be used to build a SQL SELECT. Finally, note that the class also has a method that returns the number of rows in the database table.

After the class has been added to the project, the *IndexController* class must be changed:

```
public class IndexController implements Serializable
{
 @PersistenceUnit
 EntityManagerFactory emf;
 @Resource
 UserTransaction utx;

 private Address person = new Address();
 private List<Address> persons = new ArrayList();

 public IndexController()
 {
 }

 …

 public List<Address> getPersons()
 {
  try
  {
   AddressJpaController ctrl = new AddressJpaController(utx, emf);
   persons = ctrl.findAddressEntities();
  }
```

```
  catch (Exception ex)
  {
  }
  return persons;
 }

 public void add()
 {
  try
  {
   AddressJpaController ctrl = new AddressJpaController(utx, emf);
   ctrl.create(person);
   person = new Address();
  }
  catch (Exception ex)
  {
  }
 }
}
```

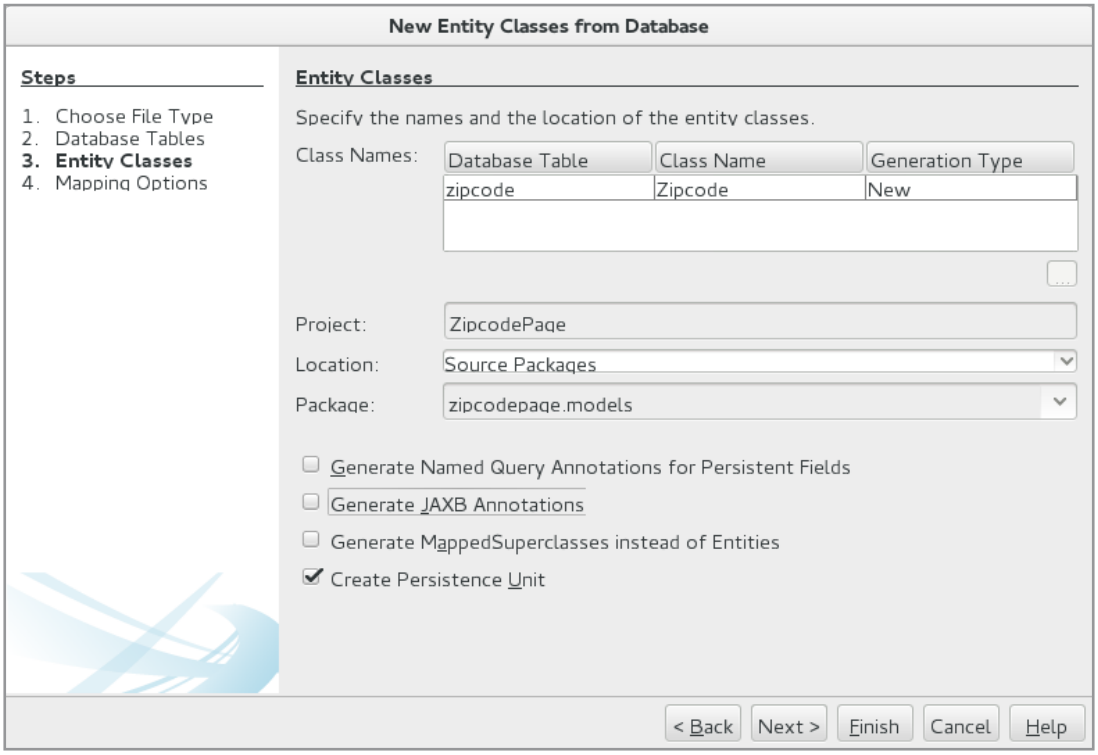
Here, the most important are the first two statements, which define an *EntityManagerFactory* and a *UserTransaction*. In particular, note the two annotations that defines the two objects to be instantiated. You should also note that the two methods that returns all addresses and create an address now have been changed so that the entire logic regarding database operations has been moved to the controller class.

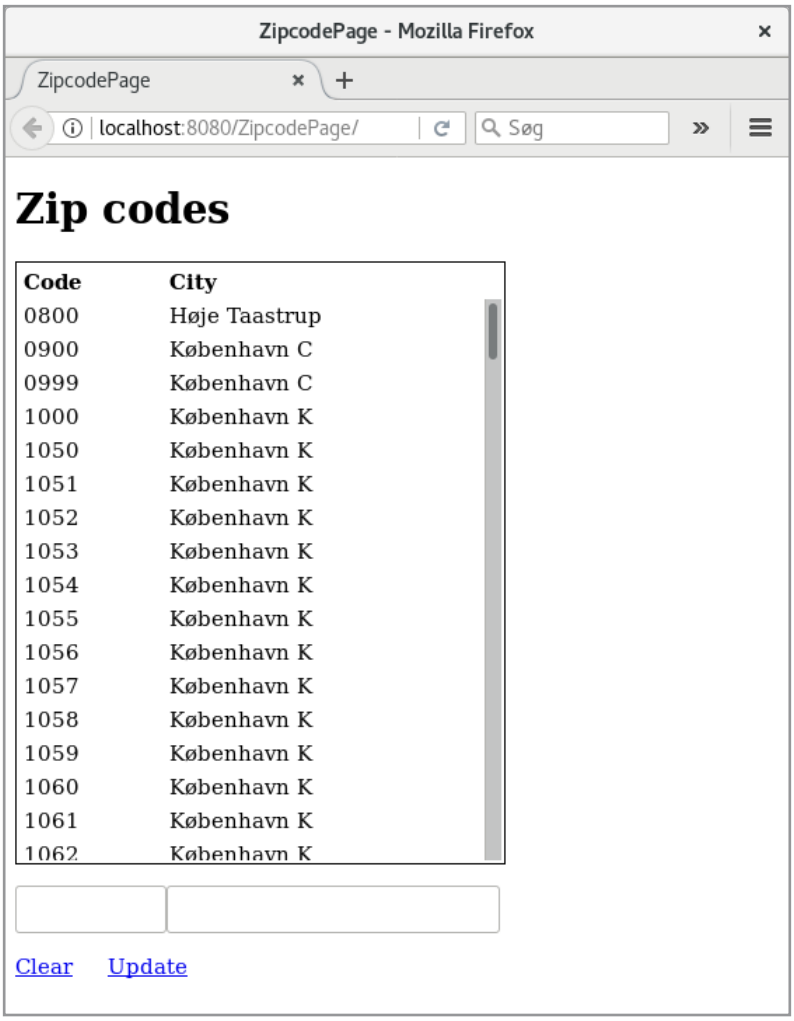
If you look at the example, you should note that everything about SQL is gone and has been moved to code that is part of the JPA API. As a result, the code to be written is significantly reduced, and in particular, you should note that the *AddressJpaController* class is autogenerated by NetBeans.

EXERCISE 1

In this exercise you must write a web application that opens and displays the content of the table *zipcode* in the database *padata* when the application has to use JPA. Start with a new web application project, which you can call *ZipcodePage*. You must then add an entity class as an *Entity Classes from Datebase*. You must define a *Data Source* for database *padata* and after you have done so, you should be able to view all tables in the database. Here you have to choose the table *zipcode*. When you get to the next window (see below), you must select a package and remove some checkmarks, but not the last checkmark. After you run the wizard, you've created an entity class named *Zipcode*, and when you examine the code, you should note that fewer named queries have been created this time. The reason is that the following window is not checked for *Generate Named Query Annotations for Persistent*

Fields. Note that a *Named Query* is an example of a SELECT statement written as a variant of SQL called *JPQL*.





As a next step, you need to add a database controller, named *ZipcodeJpaController*, for your entity class like in the previous example.

You must create a named bean for *index.xhtml* when the result must be the above window, which shows a table with the zip codes. The bottom entry fields should act as a filter:

1. the content of the first field must match all zip codes starting with the value
2. the content of the second field must match all city names that contains the value

The *Clear* link should clear the two entry fields (but should not update the table), while the *Update* link should update the table corresponding to the content of the filter. Loading the database table should only take place, when the program starts. You can do that in your bean to write a method *init()* and decorate it as shown below:

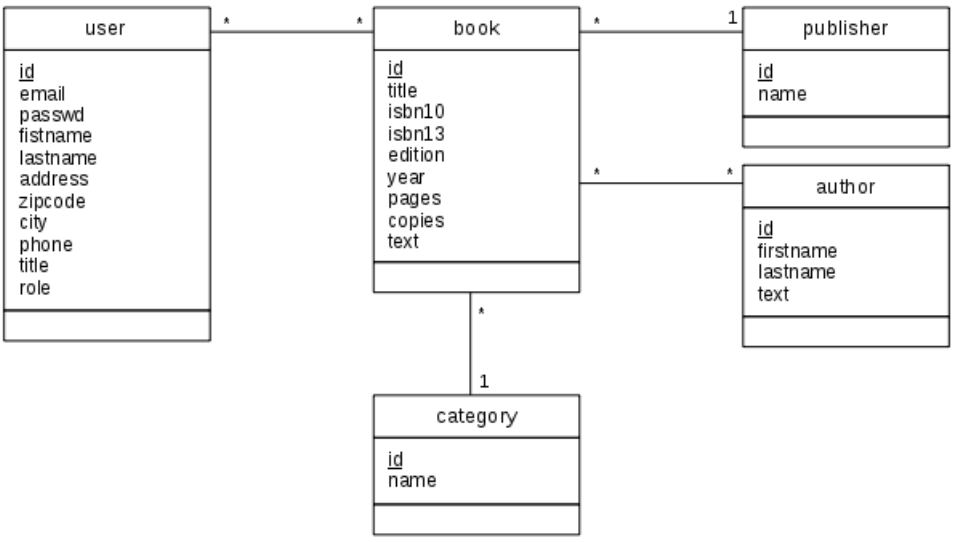
```
@PostConstruct
private void init()
{
 ZipcodeJpaController ctrl = new ZipcodeJpaController(utx, emf);
 zipcodes = ctrl.findZipcodeEntities();
}
```

It is a method that is performed immediately after the object is instantiated.

2.2 RELATED TABLES

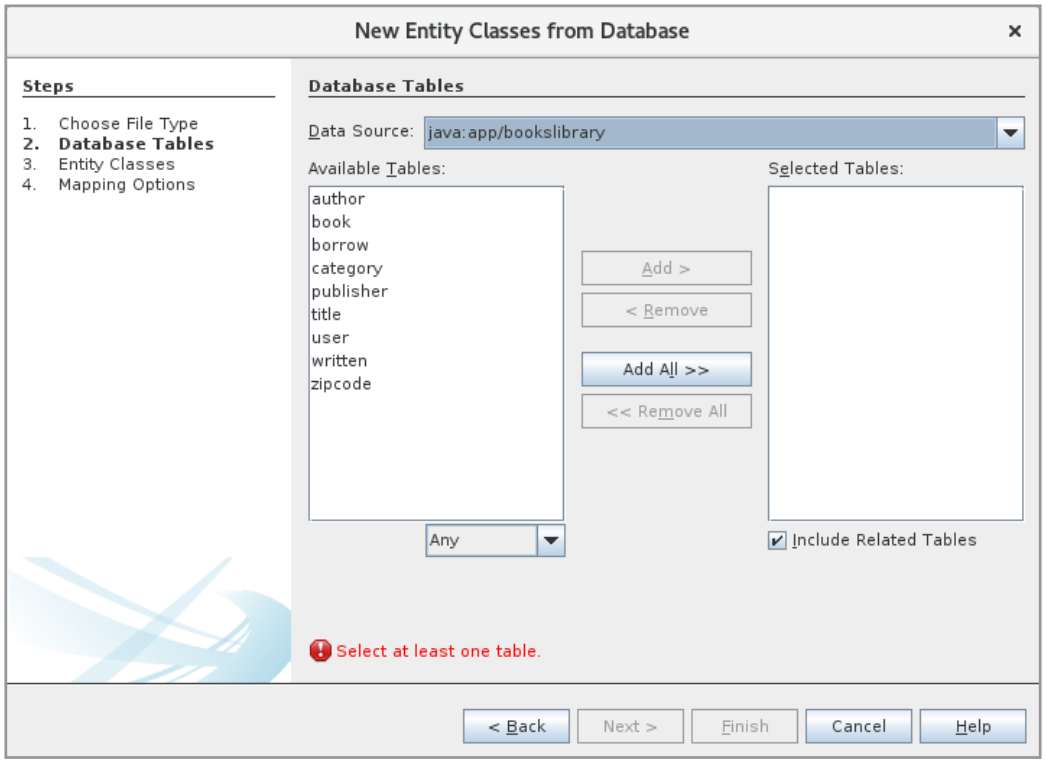
Writing a web application like the above is because of JPA quite simple, but of course it is also a simple application and primarily because the database has only one single table. In this section I will show an example where the database has more tables and where there are relationships between the individual tables, and here the benefits of JPA become even more pronounced.

As an example, I will use the Database Library from the book Java 7:



I will only use the four entities *book*, *publisher*, *category* and *author*. Here you should note that there is a many-to-one relationship between *publisher* and *book* as well as between *category* and *book*. In addition, there is a many-many relationship between *book* and *author*. The example is a web application called *PaBooks*, and it should generally be able to maintain the tables corresponding to the above four entities. It should be noted that the user interface is simple – simply to keep focus on database transactions and JPA. In the following, as in the first two examples of this chapter, I will primarily focus on how the program is made, and especially how to use a JPA to develop a database application.

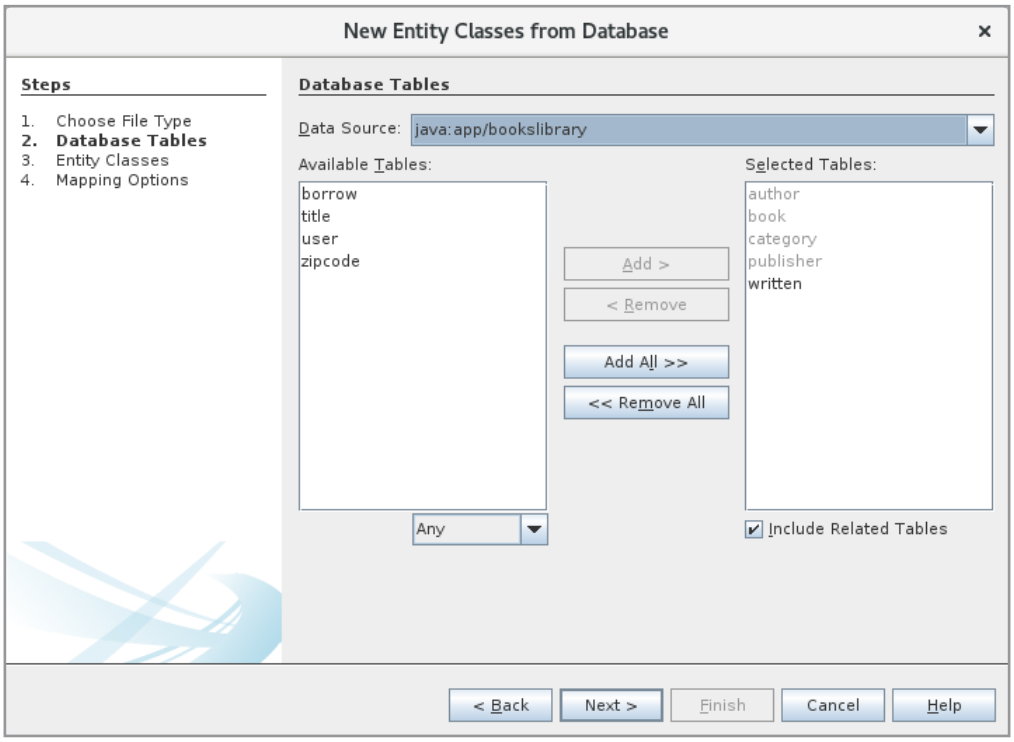
The start is a web application called *PaBooks*, and after the application has been created, I have selected *Entity Classes from Database*, and here I created a data source *booklibrary* that creates a connection to the *library* database and after doing that, you get an overview of the database tables:



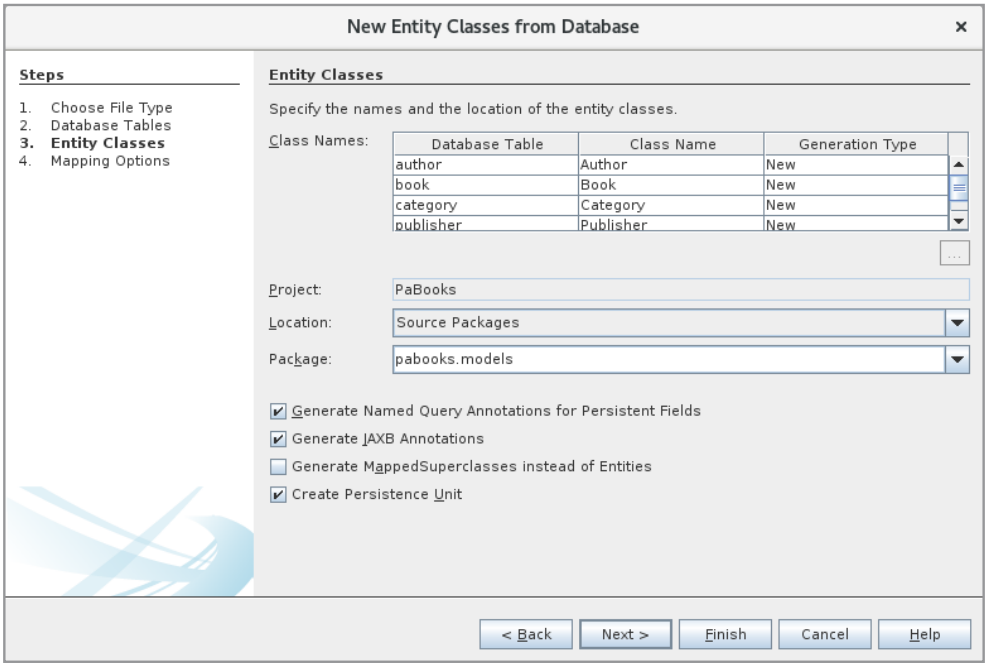
Here are the tables

- author
- book
- category
- publisher
- written

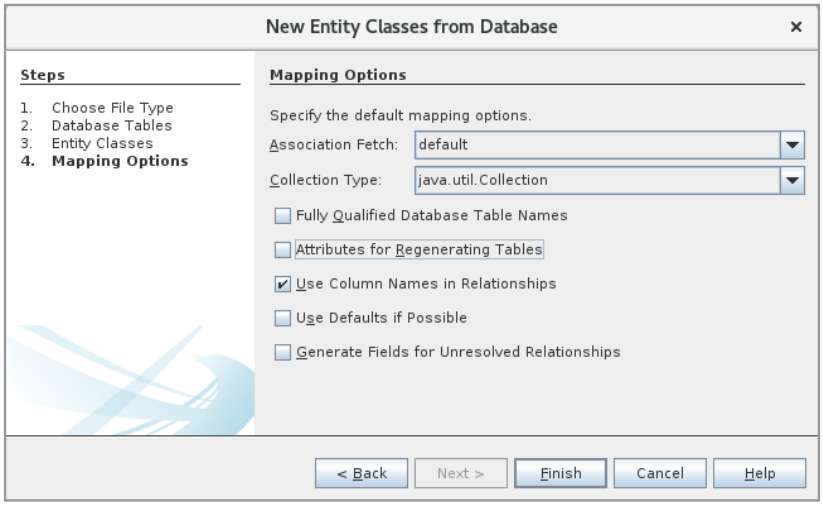
added to *Selected Tables*:



Note especially that the table *written* appears as bold, and it is because it is a relationship table between *book* and *author*. When you click *Next*, you get the following window:



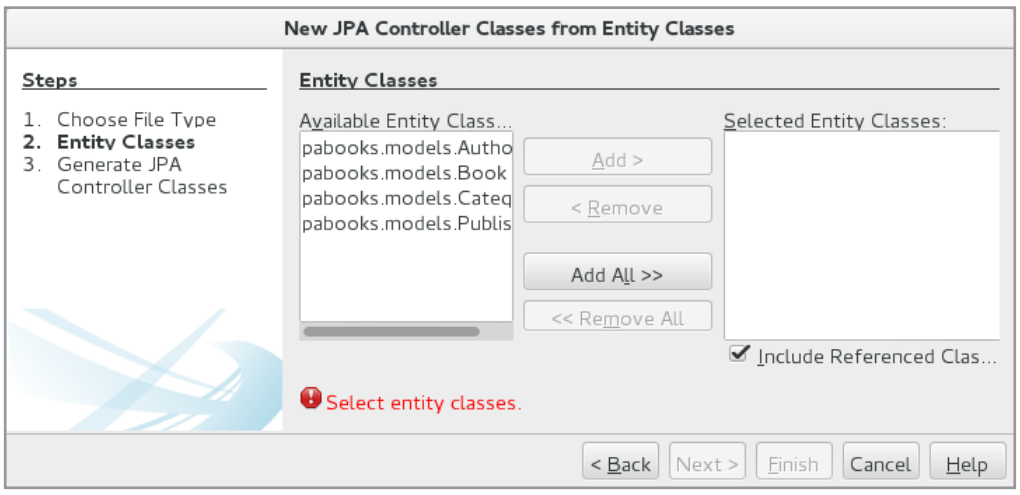
Here you can see that four entity classes are created, and note that you can change the name of those classes if you wish. I will not in this case. Also note the name of the package where the classes should be created, which I have called *pabooks.models*. When you click *Next*, you get the following window:



and when you click *Finish*, the four entity classes are created. Generally, it is rarely necessary to change these entity classes, but sometimes I change the classes' *toString()* method, which I have also done in this case for the classes *Author*, *Category* and *Publisher*. You are encouraged to study the code. Here you should note, among other things, that the *Publisher* class has a

collection *bookCollection* for *Book* objects. It represents a foreign key to *Book* objects and thus references to the *Book* objects published on this publisher. The same goes for the *Category* class. If you study the *Book* class, you should note that it has a reference to a *Publisher* object and a *Category* object, and thus the other side of the two many-to-one relationships. The *Book* class also has a collection *authorCollection* with *Author* objects that represent the many-to-many relationship between *book* and *author*. Examining the class *Author* will you see that this class has a corresponding collection *bookCollection* for *Book* objects that is the other side of the many-to-many relationship. When examining the classes, pay attention to the annotations that decorate the above properties for the relationships.

After adding entity classes, I want to add controller classes by selecting *JPA Controller Classes from Entity Classes*:



A controller must be created for all four entity classes that are added to *Selected Entity Classes*, and after clicking *Next* and selecting the desired package (here *pabooks.models*) and clicking *Finish*, the four controller classes are created:

- *AuthorJpaController*
- *BookJpaController*
- *CategoryJpaController*
- *PublisherJpaController*

Generally, these classes are not changed, however, it may be necessary to expand the classes with methods for queries on the database, and in this case, I have extended the *BookJpaController* class with a new method. However, it requires some syntax for how to write a SQL SELECT statement in JPA.

Criteria API

If you look at how I have written database applications so far, some of the most comprehensive types of typing are SQL SELECT statements – at least to write them correctly, and the same applies to JPA. To write and execute SELECT statements, you have to use a Query object, and JPA defines an SQL similar language, called *JPQL*. The language has essentially the same syntax as SQL and will not be treated explicitly in this book. When it is difficult to write SQL SELECT statements correctly, it is because it is only text and therefore the syntax can not be validated by the compiler. The problems are the same with JPQL, and to help with that, an API is defined called *Criteria API*. In fact, it is not easier to write SELECT statements using this API, but the different parts of a SELECT expression are defined using Java classes, and the compiler can thus validate if the expression is written correctly. In this book, I will use this *Criteria API* in most places, and the following is a brief introduction to the syntax.

The API is used to define criteria queries, which basically is an object graph, and you define a criteria query by building such a graph. This happens with an object of the type *CriteriaBuilder* that can create the parts that a criteria query consists of, and since it is all about representing a SELECT statement, it means object-oriented representation of tables,

columns, and conditions for which rows to extract from the database. For example, you can get a *CriteriaBuilder* from an *EntityManager* by a statement of the form:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

where *em* is an *EntityManager*. Such a builder has three methods that return a criteria query:

- *CriteriaQuery<T> createQuery(Class<T>)*
- *CriteriaQuery<Tuple> createTupleQuery()*
- *CriteriaQuery<Object> createQuery()*

and the difference is what kind of objects the query returns. After having a criteria builder that has created a query, the task is to initialize this query, and this is where the API comes on the path.

I want to start with the first of the above methods that creates a query where the parameter type indicates the expected type of objects that are the result of the query, and in principle it can be any arbitrary object type. As an example, below is shown a criteria query that determines all books in the library database, where pages have the value 520:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> cq = cb.createQuery(Book.class);
Root<Book> root = cq.from(Book.class);
cq.select(root);
cq.where(cb.equal(root.get(Book_.pages), 520));
return em.createQuery(cq).getResultList();
```

Note how to create a *CriteriaBuilder* (*em* is an *EntityManager*) and how it is used to create a criteria query for *Book* objects. For this query, a *root* is created for the object graph, and it is defined that the query object must extract objects from this root. *root* then corresponds to the FROM section in a SQL SELECT statement. Next, it is defined by the method *select()* which properties are part of the result, and here is the root which means *Book* objects. The *select()* function thus indicates which columns are to be included. Finally, it is defined that objects must only be extracted where the page number is 520, and thus corresponds to the WHERE section in a SELECT statement. The result is a collection of these objects. You should especially note how to refer to the column *pages* (the property *pages* in *Book*). The query can also be written as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> cq = cb.createQuery(Book.class);
Root<Book> root = cq.from(Book.class);
return em.createQuery(
 cq.select(root).where(cb.equal(root.get("pages"), 520))).getResultList();
```

It has no special advantages besides being a bit shorter, and so it may look more like a SQL SELECT statement. However, you must note that I have defined the column (*pages*) in another way, namely, as a string, which should be the name of that property in the *Book* class.

If you look at the code, you can notice that a criteria query object has the following methods:

- *from()*
- *select()*
- *where()*

where the meaning is known from a SELECT statement, and if you examine the type *CriteriaQuery*, you will see that there are methods for the other parts of a SELECT as ORDER BY, GROUP BY, and more. Finally, you can note that the *CriteriaBuilder* class, in addition to creating *CriteriaQuery* objects, has methods that define conditions – predicates – and many others (in the example above *equals*), where a few are illustrated below.

As another example, the query below determines the page number of all books, where the title contains the text Fedora:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Integer> cq = cb.createQuery(Integer.class);
Root<Book> root = cq.from(Book.class);
cq.select(root.get(Book_.pages));
cq.where(cb.like(root.get(Book_.title), "%Fedora%"));
List<Integer> pages = em.createQuery(cq).getResultList();
```

To compare with SQL, you can say that with *Root* you specify the table to which you want to perform a SELECT, and in connection with JPA, it means an entity object. With the *select()* method, you specify the columns or properties that will be included in the result. Here it is only the property for page numbers, and

```
Book_.pages
```

means the property *pages* in an entity object *Book*. Finally, as mentioned above, you indicate which entity objects are to be included, and here it is only those objects in which the property *title* contains the word Fedora. Note that the result is a list of *Integer* objects, and that a *CriteriaQuery* for *Integer* objects has been created from the start.

As a further example, a query is shown which determines the largest page number in the table *book*, where you should first notice that a criteria builder supports multiple functions (here the *max* function) and that the result is determined by *getSingleResult()* instead of *getResultList()*:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Integer> cq = cb.createQuery(Integer.class);
Root<Book> root = cq.from(Book.class);
cq.select(cb.max(root.get(Book_.pages)));
Integer pages = em.createQuery(cq).getSingleResult();
```

or shorter:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Integer> cq = cb.createQuery(Integer.class);
Integer pages = em.createQuery(
 cq.select(cb.max(cq.from(Book.class).get("pages")))).getSingleResult();
```

I will then show with some examples how to determine the values for specific columns, and thus specific properties in the entity class. The next query determines *id* and *title* of all *Book* objects where the page number are greater than 300:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
Root<Book> root = cq.from(Book.class);
Path<Integer> idPath = root.get(Book_.id);
Path<String> titlePath = root.get(Book_.title);
```

```
cq.select(cb.array(idPath, titlePath));
cq.where(cb.gt(root.get(Book_.pages), 300));
List<Object[]> arr = em.createQuery(cq).getResultList();
```

Here you should primarily note how to refer to the individual columns with a *Path* object. Also note how to indicate that the result should be a list whose objects are an array and here are arrays with two elements, which are defined by the method *select()*. The query can also be written as

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
Root<Book> root = cq.from(Book.class);
List<Object[]> arr = em.createQuery(
 cq.select(cb.array(root.get("id"), root.get("title"))).where(
  cb.gt(root.get("pages"), 300))).getResultList();
```

but in this case the expression is not so easy to read, and at least it requires that you get well into the *Criteria API*.

The above example is not very type strong – the result is a list of objects of the type arrays. This can be remedied with a wrapper class:

```
package pabooks.models;

public class BookWrapper
{
 private int id;
 private String title;

 public BookWrapper(int id, String title)
 {
  this.id = id;
  this.title = title;
 }

 public int getId()
 {
  return id;
 }

 public String getTitle()
 {
  return title;
 }
```

```
 public void setId(int id)
 {
  this.id = id;
 }

 public void setTitle(String title)
 {
  this.title = title;
 }
}
```

which is just a class enclosing *id* and *title* for a book. The query can then be written as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<BookWrapper> cq = cb.createQuery(BookWrapper.class);
Root<Book> root = cq.from(Book.class);
cq.select(
      cb.construct(BookWrapper.class, root.
get(Book_.id), root.get(Book_.title)));
cq.where(cb.gt(root.get(Book_.pages), 300));
List<BookWrapper> list = em.createQuery(cq).getResultList();
```

First, note that you define a criteria query for *BookWrapper* objects. Next, you use *select()* and a *construct()* method from the builder, which tells you to construct *BookWrapper* objects. The rest is in principle the same as in the previous example.

A query can also return *Tuple* objects, and for the syntax is the biggest difference that the query object is created with *createTupleQuery()*:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Book> root = cq.from(Book.class);
Path<Integer> idPath = root.get(Book_.id);
Path<String> titlePath = root.get(Book_.title);
cq.multiselect(idPath, titlePath);
cq.where(cb.gt(root.get(Book_.pages), 300));
List<Tuple> list = em.createQuery(cq).getResultList();
```

The advantage is that you do not have to write a wrapper class. Note the use of *multiselect()* to specify the columns to be included in the result, which are used to specify multiple properties in the result. Another thing is how to use the individual *Tuple* objects. I do not want to go into details here (and you are encouraged to investigate the type *Tuple* – which is an interface), but can you write the following (although the code does not make much sense in this context):

```
for (Tuple t : list)
{
 Integer id = t.get(idPath);
 String str1 = (String)t.get(1);
 String str2 = t.get(1, String.class);
}
```

A criteria query object defines a query from one or more entities (in the above examples only one that has been *Book* each time). Of course, you can specify more entities that correspond to a join operation in SQL. I will start with a query that determines all books where the name of the publisher starts with the word *Sams*:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> cq = cb.createQuery(Book.class);
Root<Book> books = cq.from(Book.class);
Join<Book, Publisher> pubs = books.join("pubid");
cq.select(books).where(cb.like(pubs.get("name"), "Sams%"));
return em.createQuery(cq).getResultList();
```

First of all, I have created a query for *Book* objects and defined a root for the entity class *Book* – which has been named *books* this time. There is a one-to-many relationship between

books and publishers, so that more books can be published on the same publisher while a book is always published on a particular publisher. I then define a *join* between *Book* and *Publisher* (two entity classes) called *pubs* by specifying the property *pubid* in the entity class *Book* as join property. The rest does not contain anything new, but you should note how to in the method *where()* to references the property *name* in the *Publisher* class using the name *pubs*.

The next example is the same join, but the query determines all books where the title contains the word *Fedora* or the publisher starts with the word *Sams*:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> query = cb.createQuery(Book.class);
Root<Book> books = query.from(Book.class);
Join<Book, Publisher> pubs = books.join("pubid");
query.select(books).where(
 cb.or(cb.like(books.get("title"), "%Fedora%"),
       cb.like(pubs.get("name"), "Sams%")));
return em.createQuery(query).getResultList();
```

You should primarily note how the the method *or()* is used to construct a condition.

There is also a one-to-many relationship between *Book* and *Category*, so you can also define a join from *Book* to *Category*. The following query determines all *Book* objects where the title contains a search value, the name of the publisher contains a search value and the name of the category contains a search value:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> query = cb.createQuery(Book.class);
Root<Book> books = query.from(Book.class);
Join<Book, Publisher> pubs = books.join("pubid");
Join<Book, Category> cats = books.join("catid");
Predicate pc = cb.and(cb.like(books.get("title"), "%" + title + "%"),
 cb.like(pubs.get("name"), "%" + pubname + "%"),
 cb.like(cats.get("name"), "%" + catname + "%"));
query.select(books).where(pc);
return em.createQuery(query).getResultList();
```

Note that two join objects are defined for the object graph. Next, a *Predicate* object is defined which represents a condition. In this case, there is an *and* of three *equal()* functions, and the predicate is used as parameters for the *where()* function. You should especially note how to use variables in search strings.

As the last example, I will show how with a query that can find all authors for a particular book. There is a many-to-many relationship between *Book* and *Author*, and that query can be written as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Author> cq = cb.createQuery(Author.class);
Root<Author> root = cq.from(Author.class);
Join<Author, Book> books = root.join("bookCollection");
cq.where(cb.equal(books.get("isbn10"), "0-672-32584-5"));
cq.select(root);
List<Author> list = em.createQuery(cq).getResultList();
```

First, a query is created for *Author* objects while *root* is defined for the entity class *Author*. Next, a join from *Author* to *Book* is defined, and with the *where()* method, it is defined that all authors who have written the book with isbn 0-672-32584-5 must be selected.

In this case (the example *PaBooks*), on the start page, I want to show an overview of all the books in the database, but such that you can filter books by searching for those books

1. where the title contains a specific search text
2. where the name of the publisher contains a specific search text
3. where the name of the category contains a specific search text

I have therefore added the following method to the *BookJpaController* class:

```
public List<Book> findBookEntities(String title, String pubname, String
catname)
{
 EntityManager em = getEntityManager();
 try
 {
  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Book> query = cb.createQuery(Book.class);
  Root<Book> books = query.from(Book.class);
  if (pubname.length() > 0 && catname.length() > 0)
  {
   Join<Book, Publisher> pubs = books.join("pubid");
   Join<Book, Category> cats = books.join("catid");
   Predicate pc = cb.and(cb.like(books.get("title"), "%" + title + "%"),
    cb.like(pubs.get("name"), "%" + pubname + "%"),
    cb.like(cats.get("name"), "%" + catname + "%"));
   query.select(books).where(pc);
  }
  else if (pubname.length() > 0)
  {
   Join<Book, Publisher> pubs = books.join("pubid");
   Predicate pc = cb.and(cb.like(books.get("title"), "%" + title + "%"),
    cb.like(pubs.get("name"), "%" + pubname + "%"));
   query.select(books).where(pc);
  }
```

```
  else if (catname.length() > 0)
  {
   Join<Book, Category> cats = books.join("catid");
   Predicate pc = cb.and(cb.like(books.get("title"), "%" + title + "%"),
    cb.like(cats.get("name"), "%" + catname + "%"));
   query.select(books).where(pc);
  }
  else
  {
   query.select(books).where(cb.like(books.get("title"), "%" + title + "%"));
  }
  return em.createQuery(query).getResultList();
 }
 catch (Exception ex)
 {
  return new ArrayList();
 }
 finally
 {
  em.close();
 }
}
```

The method may not be quite simple, but it is primarily because search texts can be blank and that the database does not necessarily contain a value for the publisher or category.

JSF Pages

With the above in place, only the individual JSF pages are returned, and although there are some details to be resolved, it basically does not contain anything new, and I would primarily refer to the completed code. If you opens the application in the browser, you get the following window (there are three books in the database):

The page has three input fields for entering search text, and furthermore there are four links that refer to pages

1. *book.xhtm*, used to create a new book
2. *publishers.xhtml*, which are used to maintain publishers
3. *categories.xhtml,* used to maintain categories
4. *authors.xhtml*, used to maintain authors

Finally, the title of each book is a link and clicked on it, you are sent to *book.xhtml* with the option of editing that book.

index.xhtml (the above window) is simple, but has a controller *IndexController*. The class is a relatively simple named bean. You should especially note how it calls the method added to the *BookJpaController* class, which performs a SQL SELECT.

The other four pages consist of a JSF page and a named bean. When studying the code, pay special attention to how individual beans implements the action methods for commands in the JSF pages and how these methods use the database controller classes. Since *book.xhtml* and its bean are not quite simple, primarily because it must be possible to maintain the many-to-many relationship between the *book* and the *author* tables. In particular, you should be aware that the *BookJpaController* class takes care of everything needed to update the database, including maintaining the relationships, both in connection with INSERT, UPDATE and DELETE. It is actually here that you meet the very great benefits of JPA, as otherwise as it appears from examples in previous books it is necessary to write a lot of code. In addition, it should be added that the JPA is even extremely effective.

PROBLEM 1

The database *padata* has three tables

1. *world*, which contains the names of the continents of this world
2. *currency*, which contains a currency table
3. *country*, which contains information about countries

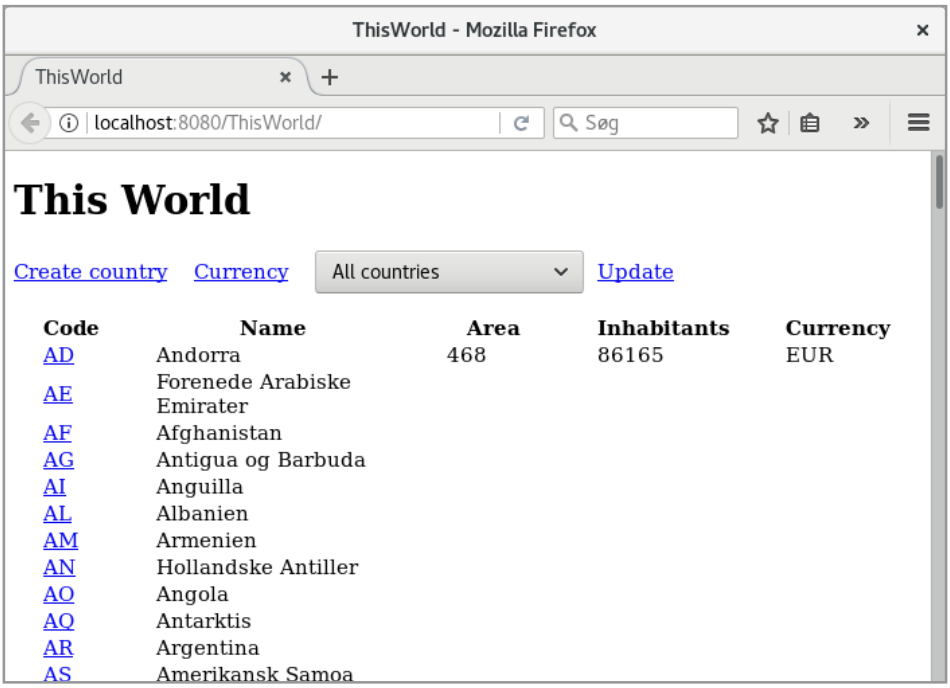
(see possible problem 1 in the book Java 6). As documentation, the databases are created using the following script:

```
create table currency
(
 code char(3) not null primary key,
 name varchar(30) not null,
 rate decimal(10, 4)
);

create table world
(
 code char(2) not null primary key,
 name varchar(15) not null
);

create table country
(
 code2 char(2) not null primary key,      # country code on 2 characters
 code3 varchar(3),                        # country code on 3 characters
 name varchar(50) not null,               # country name
 area int,                                #country's area in square kilometer
 number int,                              # country's number of inhabitant
```

```
 continent char(2),                          # the continent this country belongs
 currency char(3),                           # currency code
 foreign key (continent) references world(code),
 foreign key (currency) references currency(code)
);

insert into world values('AS', 'Asia');
insert into world values('AF', 'Africa');
insert into world values('NA', 'North America');
insert into world values('SA', 'South America');
insert into world values('AN', 'Antarctica');
insert into world values('EU', 'Europe');
insert into world values('OC', 'Oceania');
```

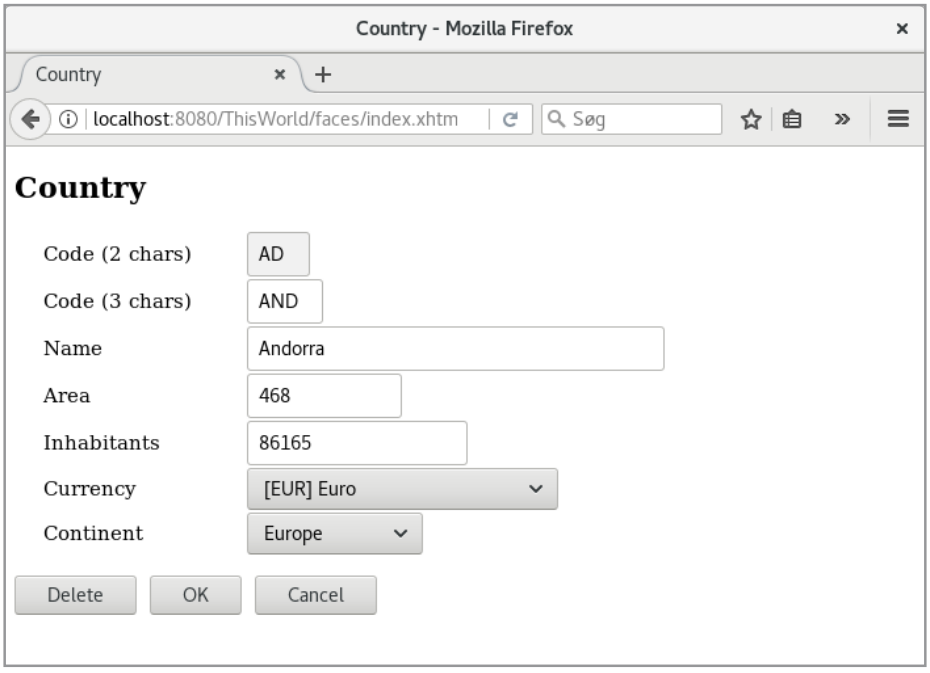
The task is to write a web application that can maintain these tables when the database operations are to be performed using JPA.

The application must have three pages. The start page should be the following with only a few countries shown:

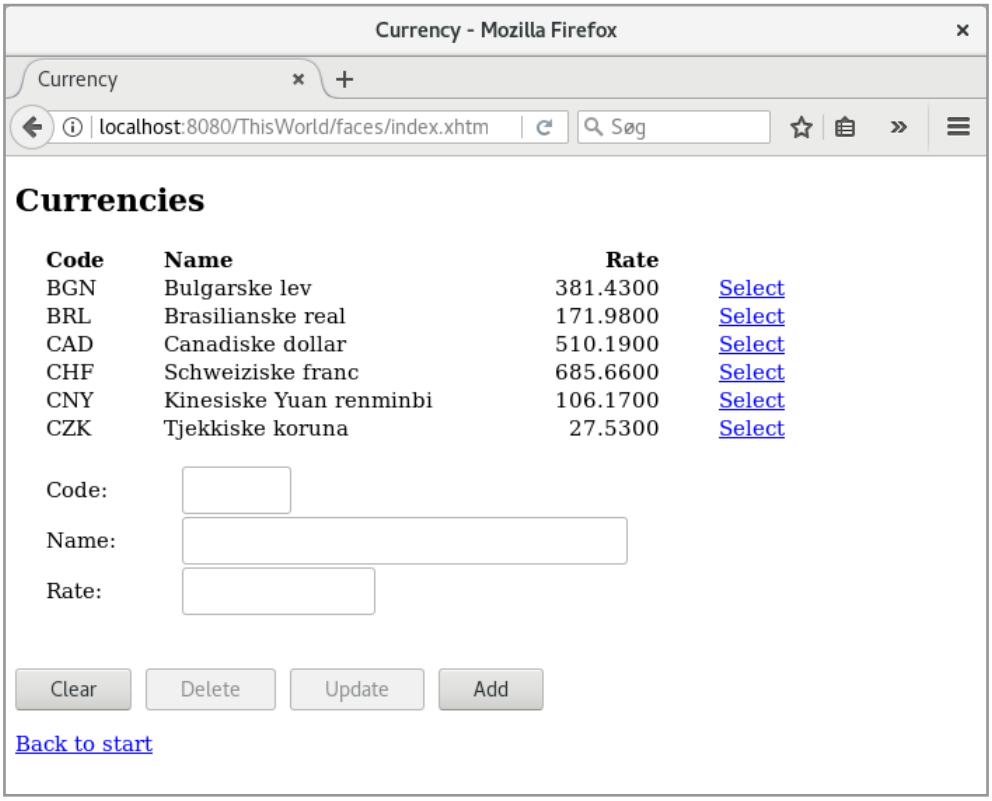


At the top there is a link to a page that is used to create a country – the same page is used to edit information about a country if you click on the country code in the list. The link *Currency* is used to maintain the currency table. Finally, there is a dropdown box where you can select a continent, and then clicking on the *Update* link the list is updated so that it only shows countries for the selected continent.

The page for maintenance information about countries is the following where the code for *Andora* is clicked:



Finally, the page for the maintenance of the currency table is shown below (where only a few currencies are shown):



and the page should be self explanatory.

3 ENTERPRISE JAVA BEANS

Entreprise Java Beans or EJB's are components that live on a server and enclose business logic, and here they can be used by other applications such as web applications and standalone applications. You could thus think of an EJB as a component that is developed, tested and hosted on an application server (such as Glassfish), and as other applications may apply. It is not quite simple to develop EJB's and get them hosted on an application server, and the architecture is also not so easy to review, but NetBeans helps as it has projects specifically aimed for developing EJB's. In this chapter, I will partly demonstrate how to develop and use EJB's using NetBeans and explain the most important concepts.

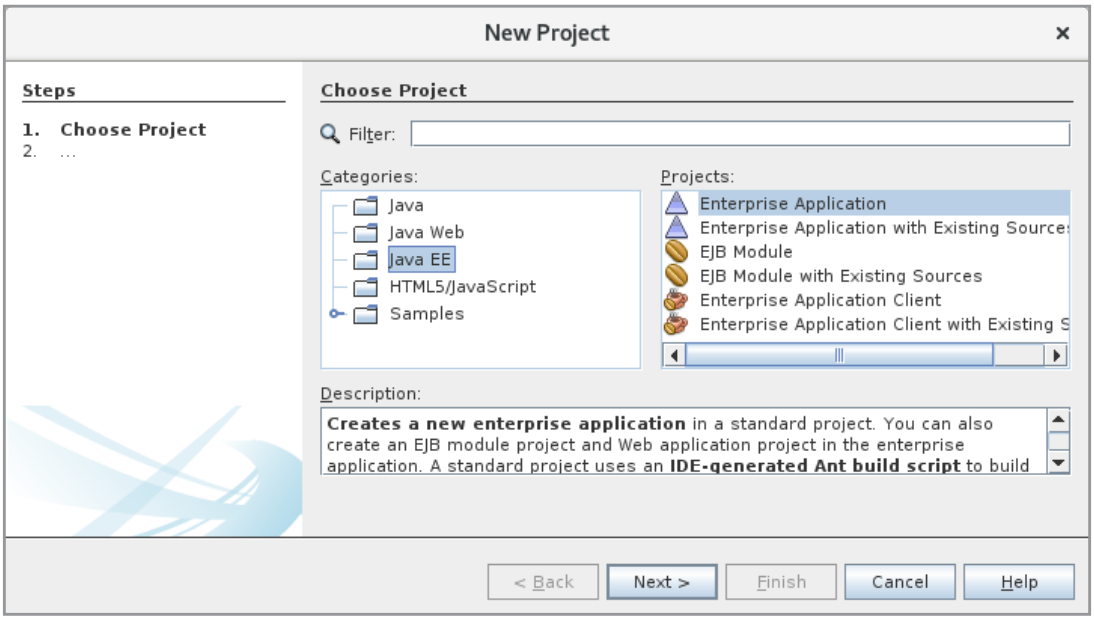
The advantage of using EJB's is that, in addition to providing the same services to multiple applications, the application server addresses a variety of conditions such as transactions and security issues, and EJB components can thus simplify the development of an enterprise application. EJB's therefore play an important role in the development of modern applications, including distributed applications, but it should be mentioned immediately that they are no longer playing the same role because of web services.

There are three variants:

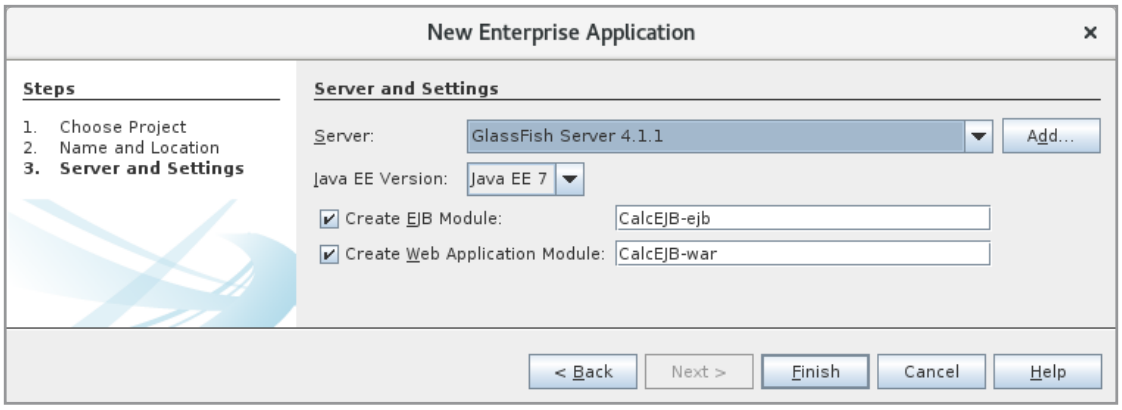
- *Stateless*, which are components where the state is not preserved between several calls to the component's methods.
- *Stateful*, that are components where the state is maintained throughout the life of the component as compared to the current application.
- *Singleton*, which is a component that exists in exactly one version relative to an application.

Having said that it is not easy to write EJB modules, it is not so much to write the individual EJB classes, as that are no different from writing other classes, but the problems arise in connection with deployment and the use of the modules from client programs, and thus relates more to Glassfish than to writing the code. Therefore, I will start simple and create an *Enterprise Application* project, consisting of a single EJB module, as well as a web application that uses this module. The EJB module should consist of a single stateless session bean that provides five methods (services) available to clients that here are the web application.

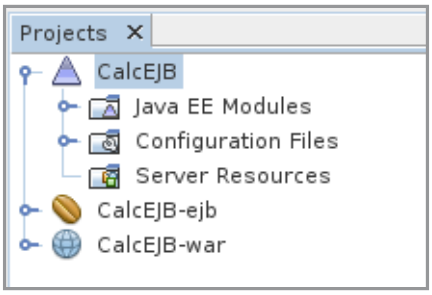
In the following I will describe the procedure. I start with a new project by selecting *Java EE and Enterprise Application*:



After clicking *Next*, I will enter the project name in the next window where I have entered *CalcEJB*. After clicking *Next* again, I get the following window, which by default consists of a single EJB module and a Web Application:

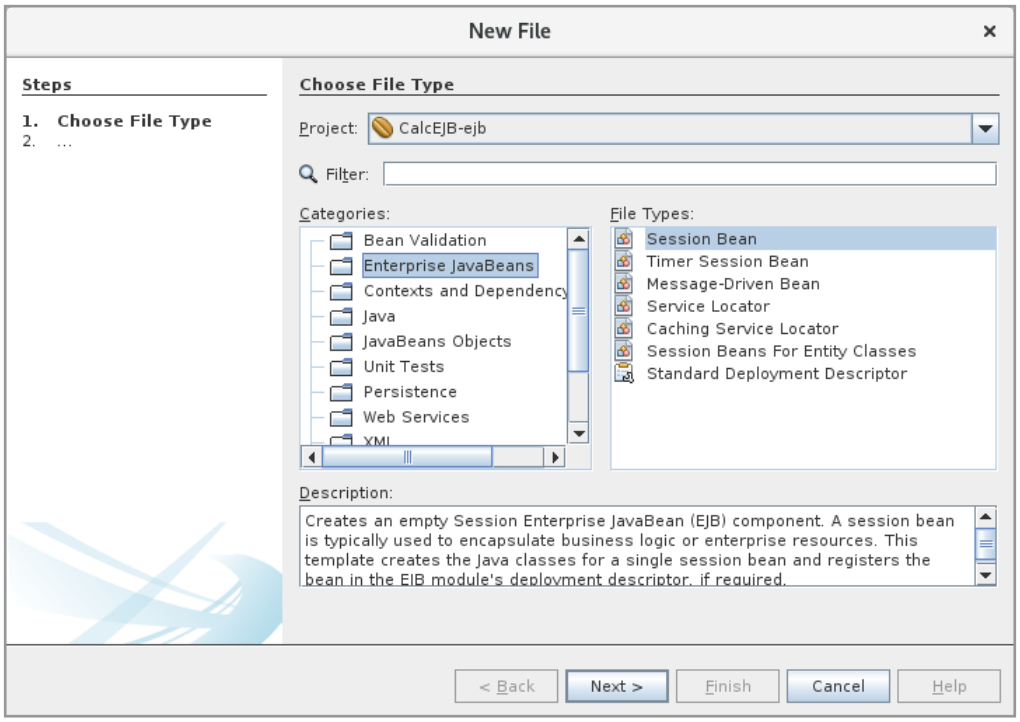


When you click Finish, NetBeans creates an Enterprise Application, which has attached two projects:

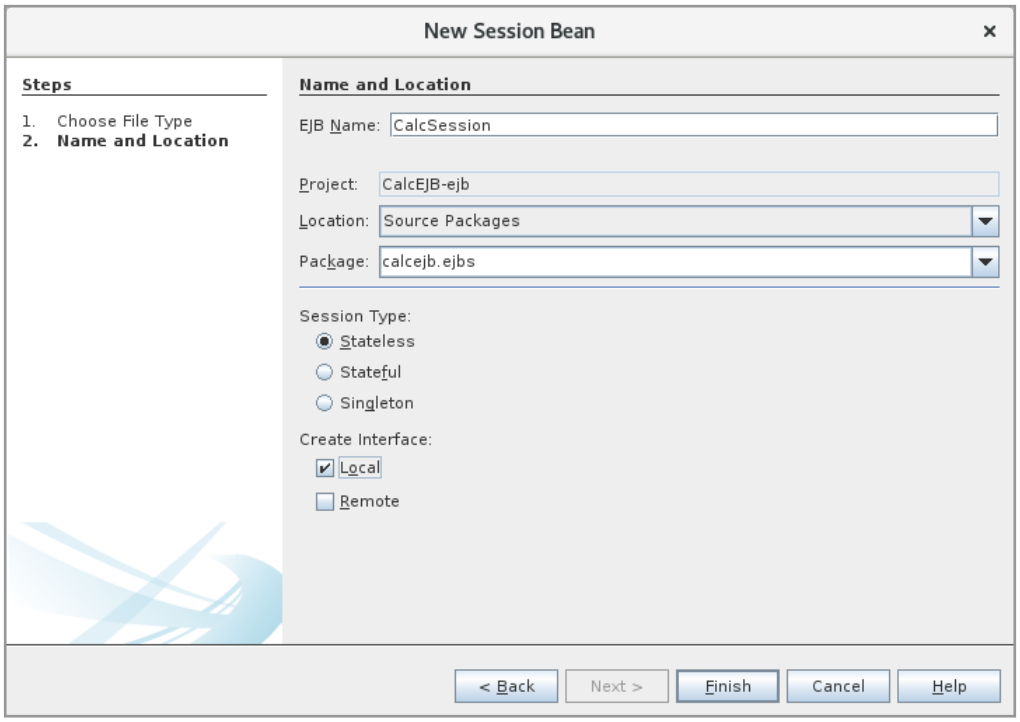


where the first one is an *EJB Module* project named *CalcEJB-ejb*, while the other is a *Web Application* project called *CalcEJB-war*. Note that it is NetBeans who has chosen the names and they can of course be changed if desired.

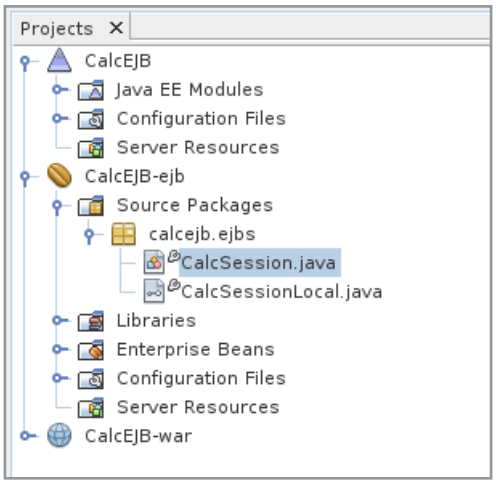
As a next step, the *CalcEJB-ejb* project should have an EJB. To do this, right-click on the project and select *Enterprise Javabeans* and then *Session Bean*:



When you clicks *Next* you get the following window, where you must enter a name for your EJB (here *CalcSession*) and a name on a package (here *calcejb.ejbs*):



In addition, choose a *Session Type* where *Stateless* has been selected. Finally, *Local* has been ticked. This means that an interface that defines the EJB is created. When here selected *Local*, it means that it is an EJB that can be used by a web applications hosted on the same application server (what will often be the case in practice) and, precisely, it means an application performed by the same JVM as the EJB module. After clicking *Finish*, NetBeans creates a class and interface:



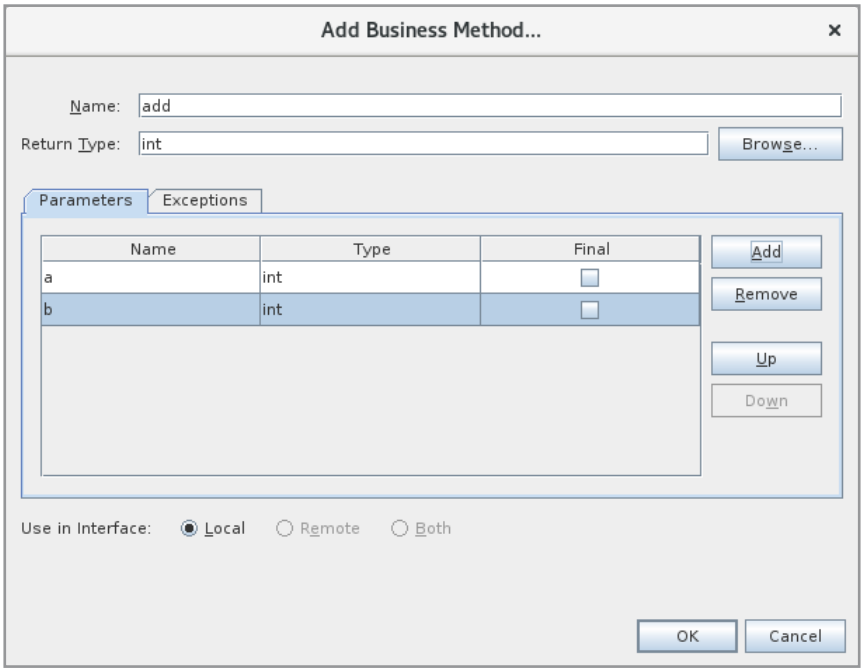
The class is as follows:

```
package calcejb.ejbs;

import javax.ejb.Stateless;

@Stateless
public class CalcSession implements CalcSessionLocal
{
}
```

and you can partly see that it implements the interface and that it is decorated with an annotation telling that it is a stateless session bean. The interface is an empty interface but decorated with an annotation that tells you that it is a local session bean:

```
package calcejb.ejbs;

import javax.ejb.Local;

@Local
public interface CalcSessionLocal
{
}
```

Next, methods for the EJB class should be added. To do this, right-click on the class name and choose *Insert Code*, and in the follow-up menu, choose *Add Business Method*, after which you get the following window:



Hereafter, NetBeans adds a method to the *CalcSession* class. The advantage of using the above tool is that it also updates the interface, which, of course, can also be done manually. Below is the completed interface that defines 5 methods:

```
package calcejb.ejbs;

import javax.ejb.Local;

@Local
public interface CalcSessionLocal
{
 int add(int a, int b);
 int sub(int a, int b);
 int mul(int a, int b);
 int div(int a, int b);
 int mod(int a, int b);
}
```

end below the finished class, which implements all of the interface's methods:

```
package calcejb.ejbs;

import javax.ejb.Stateless;

@Stateless
public class CalcSession implements CalcSessionLocal
{
 @Override
 public int add(int a, int b)
 {
  return a + b;
 }

 @Override
 public int sub(int a, int b)
 {
  return a - b;
 }

 @Override
 public int mul(int a, int b)
 {
  return a * b;
 }

 @Override
 public int div(int a, int b)
 {
  try
  {
   return a / b;
  }
  catch (Exception ex)
  {
   return 0;
  }
 }

 @Override
 public int mod(int a, int b)
 {
  try
  {
   return a % b;
  }
```

```
  catch (Exception ex)
  {
   return 0;
  }
 }
}
```

Now my session bean is complete, and then the client must be written. First, I have added a named bean called *CalcBean* (to the package *calcejb.beans*). After I've added the class, I have right-clicked on the class name and selected *Insert Code* and here again *Call Enterprise Bean*, after which the result is:

```
package calcejb.beans;

import calcejb.ejbs.CalcSessionLocal;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
import javax.ejb.EJB;

@Named(value = "calcBean")
@SessionScoped
public class CalcBean implements Serializable
{
 @EJB
 private CalcSessionLocal calcSession;
 public CalcBean()
 {
 }
}
```

That is, a reference has been made to the EJB module, and the important thing here is that it is decorated with *@EJB*. This means that the relevant session bean can be used by the class's methods. The finished name bean is as follows:

```
package calcejb.beans;

import calcejb.ejbs.CalcSessionLocal;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
import javax.ejb.EJB;

@Named(value = "calcBean")
@SessionScoped
```

```
public class CalcBean implements Serializable
{
 @EJB
 private CalcSessionLocal calcSession;

 private int value1;
 private int value2;
 private int value3;

 public CalcBean()
 {
 }

 public int getValue1() {
  return value1;
 }

 public void setValue1(int value1) {
  this.value1 = value1;
 }
```

```
 public int getValue2() {
  return value2;
 }

 public void setValue2(int value2) {
  this.value2 = value2;
 }

 public int getValue3() {
  return value3;
 }

 public void setValue3(int value3) {
  this.value3 = value3;
 }

 public void clear()
 {
  value1 = 0;
  value2 = 0;
  value3 = 0;
 }

 public void addition()
 {
  value3 = calcSession.add(value1, value2);
 }

 public void subtract()
 {
  value3 = calcSession.sub(value1, value2);
 }

 public void multiply()
 {
  value3 = calcSession.mul(value1, value2);
 }

 public void divide()
 {
  value3 = calcSession.div(value1, value2);
 }

 public void modolus()
 {
  value3 = calcSession.mod(value1, value2);
 }
}
```

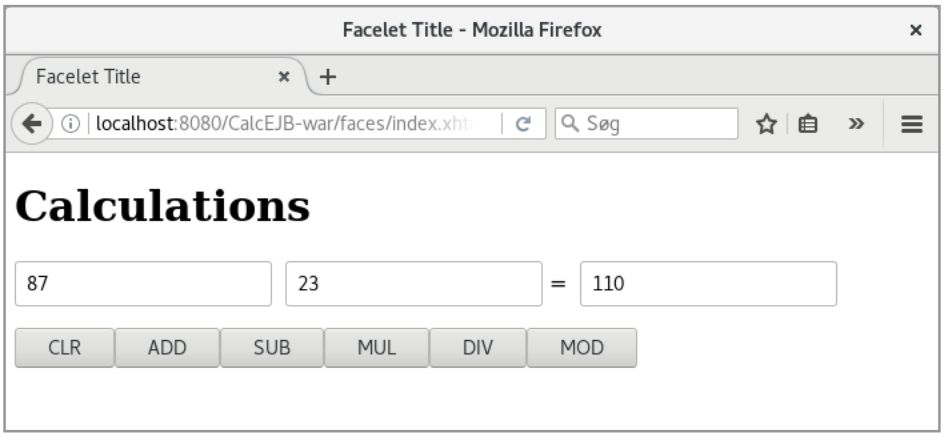
and the code does not require further explanations. However, note how the class's methods refer to the *calcSession* object, as it was any other object.

I have then deleted the page *index.html* and created a JSF page with the same name (*index.xhtml*) and I have entered the following code:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html … >
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  <h1>Calculations</h1>
  <h:form>
   <p>
    <h:inputText value="#{calcBean.value1}" /> 
    <h:inputText value="#{calcBean.value2}" /> = 
    <h:inputText value="#{calcBean.value3}" />
   </p>
   <p>
    <h:commandButton value="CLR" action="#{calcBean.clear()}" />
    <h:commandButton value="ADD" action="#{calcBean.addition()}" />
    <h:commandButton value="SUB" action="#{calcBean.subtract()}" />
    <h:commandButton value="MUL" action="#{calcBean.multiply()}" />
    <h:commandButton value="DIV" action="#{calcBean.divide()}" />
    <h:commandButton value="MOD" action="#{calcBean.modolus()}" />
   </p>
  </h:form>
 </h:body>
</html>
```

Here's nothing new to explain and you should notice that my named bean is used in the usual way.

After the code is written, you must build the projects. Then right-click on the name for enterprice application *CalcEJB* and select deploy. If you right-click again and click *Run*, you get the following window:



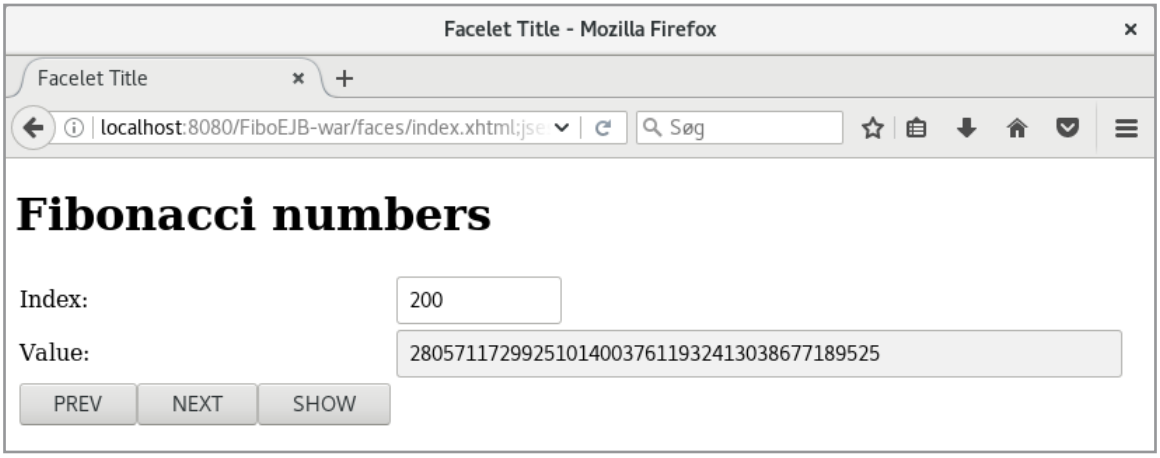
where values are entered in the first two input fields and the clicked on ADD.

3.1 A STATEFUL SESSION BEAN

In this section I will show an example of a *stateful* session bean. In fact, it is developed precisely in the same way as above, but it retains the value of instance variables between the individual calls of methods. When a *stateless* or *stateful* session bean is hosted on the

server (deployed), the server will create a number of instances of the session bean, which is commonly referred to as an EJB pool. When a client wants to use an instance of an EJB, the Glassfish server uses an instance from the pool and gives the client a reference. The goal of this process is of course performance, as it takes a relatively long time to create an instance of an EJB. Is it a stateful session bean, the server maintains a conversation with the client, so it constantly uses the same instance. This is not the case with a stateless session bean, and for every call of a method there is no guarantee that it is the same instance that is used. Immediately, it would seem beneficial to use stateful session beans, but for the server there is an overhead to maintain the state, and therefore should stateful session beans only be used if there is a need.

The current example opens the following window where is 200 entered as the index and the clicked on *SHOW*:



The fibonacci numbers are as known

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

and indexes from 0, the above window shows the fibonacci number with index 200. The *PREV* and *NEXT* buttons are used to change a number back or forth.

To write the program, I have started with an *Enterprise Application* project named *FiboEJB* and said that project must contain an EJB Module and a Web Application – just the same as in the first project. Next to the EJB module – *FiboEJB-ejb* – is added a session bean called *FiboSession*. Also, it happens as in the first project with only one exception that I have saying it should be a *stateful* session bean, but otherwise, as in the first example, it should have a local interface.

I have then written the interface, but this time I've done it manually without using NetBeans to create the necessary methods:

```
package fiboejb.ejbs;

import javax.ejb.Local;

@Local
public interface FiboSessionLocal
{
 int getIndex();
 void setIndex(int index) throws Exception;
 Object getValue();
 void next();
 void prev() throws Exception;
}
```

The meaning of the individual methods should be explained in the explanation of the above browser window, but you should note that two of the methods can raise an exception and that the method *getValue()* returns an *Object* – the method returns the current fibonacci number.

The implementation is the following, where the code is written manually:

```
package fiboejb.ejbs;

import javax.ejb.Stateful;
import java.math.BigInteger;

@Stateful
public class FiboSession implements FiboSessionLocal
{
 private int index = 0;
 private BigInteger value1 = BigInteger.ZERO;
 private BigInteger value2 = BigInteger.ZERO;

 @Override
 public int getIndex()
 {
  return index;
 }

 @Override
 public void setIndex(int index) throws Exception
 {
  if (index < 0) throw new Exception("Index must be none negative");
  while (this.index > index) prev();
```

```
 while (this.index < index) next();
}

@Override
public Object getValue()
{
 return value2;
}

@Override
public void next()
{
 if (index == 0)
 {
  value1 = BigInteger.ZERO;
  value2 = BigInteger.ONE;
  index = 1;
 }
 else
 {
  BigInteger value3 = value1.add(value2);
  value1 = value2;
  value2 = value3;
```

```
   ++index;
  }
 }

 @Override
 public void prev() throws Exception
 {
  if (index == 0) throw new Exception("Index is 0");
  if (index == 1)
  {
   value2 = BigInteger.ZERO;
   index = 0;
  }
  else
  {
   BigInteger value3 = value2.subtract(value1);
   value2 = value1;
   value1 = value3;
   --index;
  }
 }
}
```

First, note that an annotation indicates that it is a *stateful* session bean. Otherwise, the code should be easy enough to understand, but in addition to showing an example of a *stateful* session bean, the goal is to show a bean that uses a not simple type (here *BigInteger*), as well as an example of non trivial algorithms.

Then the session bean in question is finished. To write the client I have started with a named bean that uses the above session bean. I have created a named bean named *FiboBean*, and then I have used NetBeans to create a reference to an EJB instance – just the same as in the first example. The finished bean is shown below:

```
package fiboejb.beans;

import fiboejb.ejbs.FiboSessionLocal;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
import javax.ejb.EJB;
import java.math.*;

@Named(value = "fiboBean")
@SessionScoped
public class FiboBean implements Serializable
```

```
 @EJB
 private FiboSessionLocal fiboSession;

 public FiboBean()
 {
 }

 public int getIndex()
 {
  return fiboSession.getIndex();
 }

 public void setIndex(int index) throws Exception
 {
  fiboSession.setIndex(index);
 }

 public BigInteger getValue()
 {
  return (BigInteger)fiboSession.getValue();
 }

 public void next()
 {
  fiboSession.next();
 }

 public void prev() throws Exception
 {
  fiboSession.prev();
 }
}
```

The class is simple and consists primarily of methods that delegate work to the EJB instance. The example shows in many ways the use of enterprice java beans as objects that encapsulate business logic and move it away from named beans used as controller for the user interface. In this way, you simplify the individual named beans.

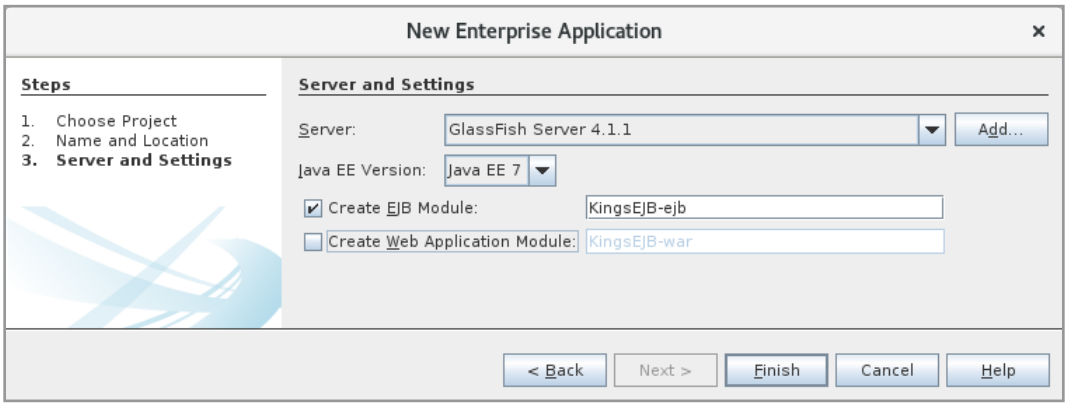
Finally, there is the code for *index.xhtml*, which contains no new, and which I do not want to show here.

3.2 A REMOTE SINGLETON SESSION BEAN

In the first two examples, both the session bean and the client application have been part of the same enterprise application. In this section I will show an example of a session bean developed as single module and deployed independently to the server. Next, the example will include a common web application that is developed independently of the above bean, but uses the session bean. This means that the application and the bean could be hosted on each application server.

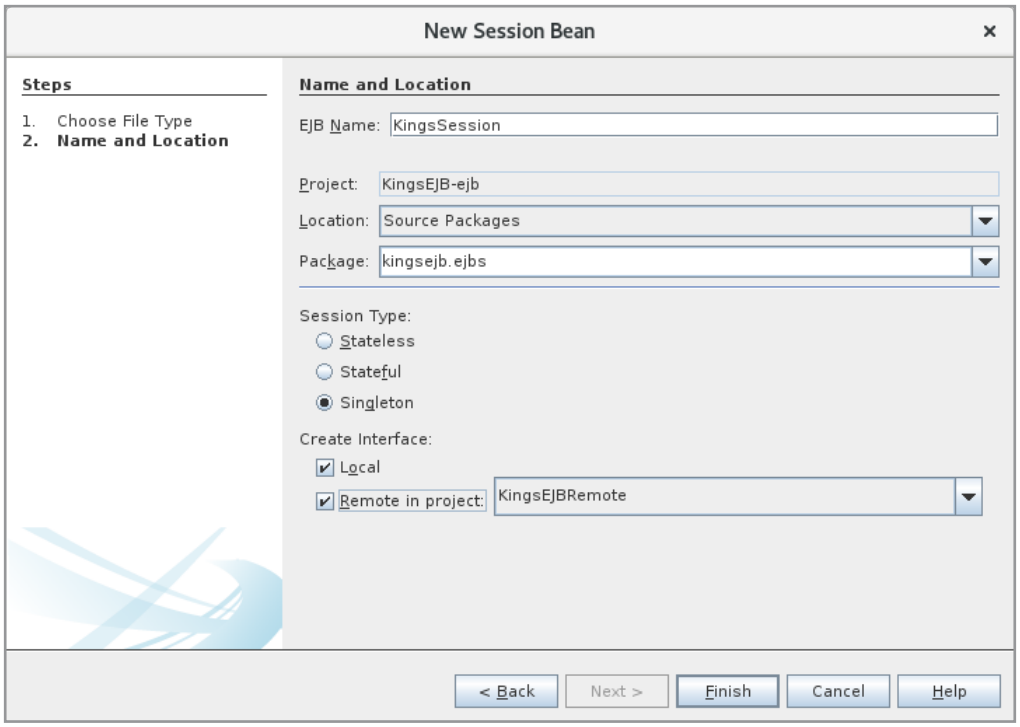
The actual session bean should also be written as a *singleton* – primarily to show how. A *singleton session bean* is a session bean, where there is exactly one instance per. application server. A typical application is a bean that must initialize some data exactly once, which should otherwise be available throughout the entire life of the application. Often, the data in question is initialized from a database, and you want to decorate a method with @*PostConstruct*, where data can be initialized. It is not currently used in the present case as data is hard-coded to make the current bean simple.

In the same way as in the previous two examples I start of creating an *Enterprise Application* project named *KingsEJB*, and when I get to the *Server and Settings* window, I have cleared the *Create Web Application Module*:

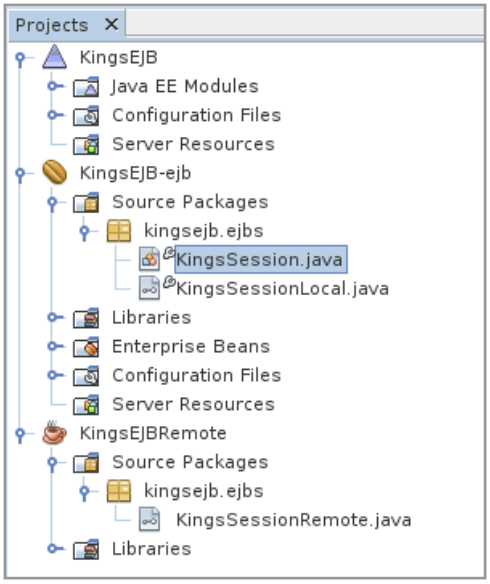


When you click *Finish*, NetBeans creates an enterprise application, but only with an EJB module. This time there must be remote access to the EJB, and it must therefore be defined with an interface. As a next step, I have therefore created a conventional *Class Library* project called *KingsEJBRemote*.

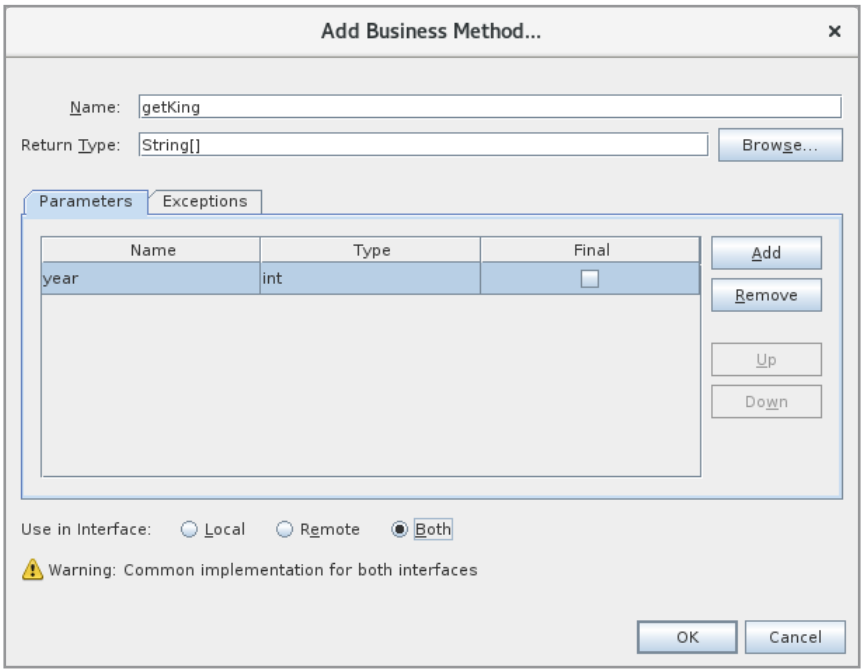
An enterprise java bean must now be written, and it must have two services, both returning an array of strings. The first must return the name of the Danish kings who ruled the year in question (there may be two kings if the year is the year of a change of faith) on the basis of a year. The other must work in the same way, but there must be two years as parameters, and the method must return the names of all kings that have ruled within that period. To create this bean, I have right-clicked the *KingsEJB-ejb* module and selected *Session Bean*. You will then get the following window where I have entered the name *KingsSession* as the package name *kingsejb.ejbs*. In addition, I have chosen it to be a *Singleton*. Finally, it has been stated that both a Local and a Remote Interface must be created. When checking *Remote*, select the class library that will contain the interface, and in this case it is *KingsEJBRemote*. For the current example, there is no need for a *Local* interface, and when I have chosen it, it is only to show that a session bean can have both a *Local* and a *Remote* Interface.



After clicking *Finish*, NetBeans has created the desired session bean as well as both interfaces:



Next, two methods should be added. I have right-clicked the name of the class *KingsSession* and selected *Insert Code* and here again selected *Add Business Method* (see the window below). The first method must be called *getKing*, and should return an array of strings and have a single parameter of the type int. At the bottom I clicked on the last radio button, which says that both interfaces should be used. You should note that NetBeans provides a warning that the method will get the same name in the two interfaces.



When you click *OK*, NetBeans will create the method and insert a prototype into both interfaces. For the sake of the above warning I have renamed them to respectively

getKingLocal(int year)
getKingRemote(int year)

and the consequence is that the *KingsSession* class must implement both methods. The other service must be defined accordingly, and as an example, I have shown the *KingsSessionRemote* interface below:

```
package kingsejb.ejbs;

import javax.ejb.Remote;

@Remote
public interface KingsSessionRemote
{
 String[] getKingRemote(int year);
 String[] getKingsRemote(int year1, int year2);
}
```

The methods must be implemented in the *KingsSession* class, where I have only shown a few data lines:

```
package kingsejb.ejbs;

import javax.ejb.Singleton;
import java.util.*;

@Singleton
public class KingsSession implements KingsSessionRemote, KingsSessionLocal
{
 @Override
 public String[] getKingRemote(int year)
 {
  return getKing(year);
 }

 @Override
 public String[] getKingsRemote(int year1, int year2)
 {
  return getKings(year1, year2);
  }

 @Override
 public String[] getKingLocal(int year)
 {
  return getKing(year);
 }
```

```
 @Override
 public String[] getKingsLocal(int year1, int year2)
 {
  return getKings(year1, year2);
 }

 private String[] getKing(int year)
 {
  List<String> list = new ArrayList();
  for (String[] king : names)
  {
   int a = king[1].length() > 0 ? Integer.parseInt(king[1]) : 0;
   int b = king[2].length() > 0 ? Integer.parseInt(king[2]) : 9999;
   if (a <= year && year <= b) list.add(king[0]);
  }
  String[] arr = new String[list.size()];
  return list.toArray(arr);
 }

 private String[] getKings(int year1, int year2)
 {
  List<String> list = new ArrayList();
  for (String[] king : names)
  {
   int a = king[1].length() > 0 ? Integer.parseInt(king[1]) : 0;
   int b = king[2].length() > 0 ? Integer.parseInt(king[2]) : 9999;
   if (year2 >= a && year1 <= b) list.add(king[0]);
  }
  String[] arr = new String[list.size()];
  return list.toArray(arr);
 }

 private static String[][] names = {
  { "Gorm den Gamle", "", "958" },
  { "Harald Blåtand", "958", "986" },
  { "Svend Tveskæg", "986", "1014" },
  …
  { "Margrethe d. 2.", "1972", "" }
 };
}
```

There is nothing new about this session bean except that it is defined as a *Singleton* session bean. Note, however, that the class implements both interfaces, thus implementing all four methods.

The EJB module is complete and can be translated and then it can be deployed to the server.

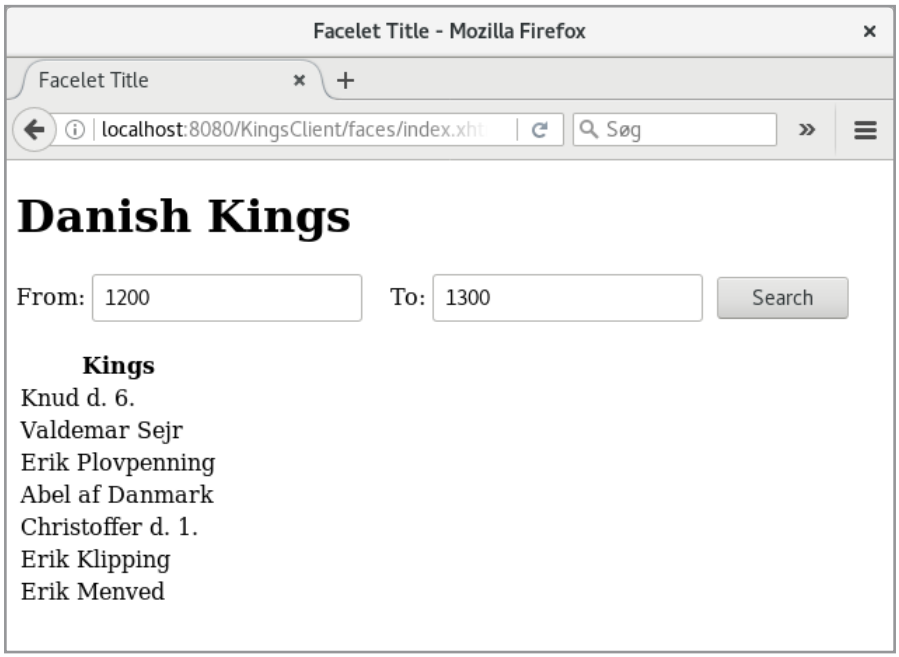
Then there is the client application, where I have created a usual Web Application project named *KingsClient*. In order for the application to refer to *KingsSession*, it must know the definition of the *KingsSessionRemote* interface. I have therefore added the jar file with this interface to the project. As the next step I have defined a named bean:

```
package kingsclient.beans;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.util.*;
import javax.ejb.EJB;
import kingsejb.ejbs.KingsSessionRemote;

@Named(value = "kingsBean")
@SessionScoped
public class KingsBean implements java.io.Serializable
{
 @EJB
 private KingsSessionRemote kingsSession;
```

```
 private int year1;
 private int year2;
 private List<String> names = new ArrayList();

 public KingsBean() {
 }

 public int getYear1() {
  return year1;
 }

 public void setYear1(int year1) {
  this.year1 = year1;
 }

 public int getYear2() {
  return year2;
 }

 public void setYear2(int year2) {
  this.year2 = year2;
 }

 public List<String> getNames()
 {
  return names;
 }

 public void search()
 {
  String[] arr = year2 > 0 ? kingsSession.getKingsRemote(year1, year2) :
    kingsSession.getKingRemote(year1);
  names.clear();
  for (String name : arr) names.add(name);
 }
}
```

Also, here is not much news, but note that the class uses the EJB module *KingsEJB*. It happens as in the previous examples by right-clicking on the name of the class, choosing *Insert Code*, and then *Call Enterprise Bean*. The relevant bean is used in the method *Search()*.

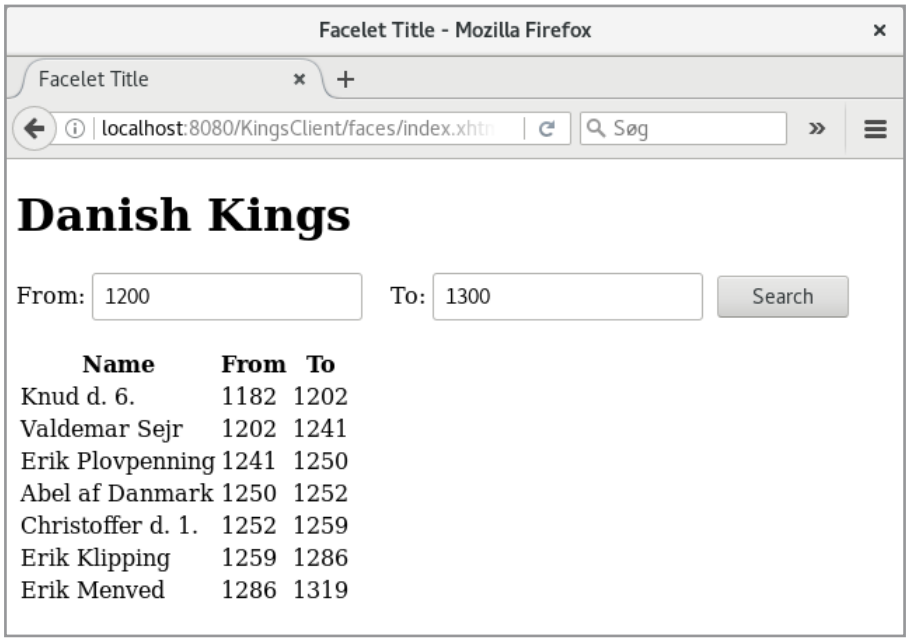
If you perform the application (opens it in the browser), you get the following window, which has been applied to all kings ruling from 1200 to 1300:



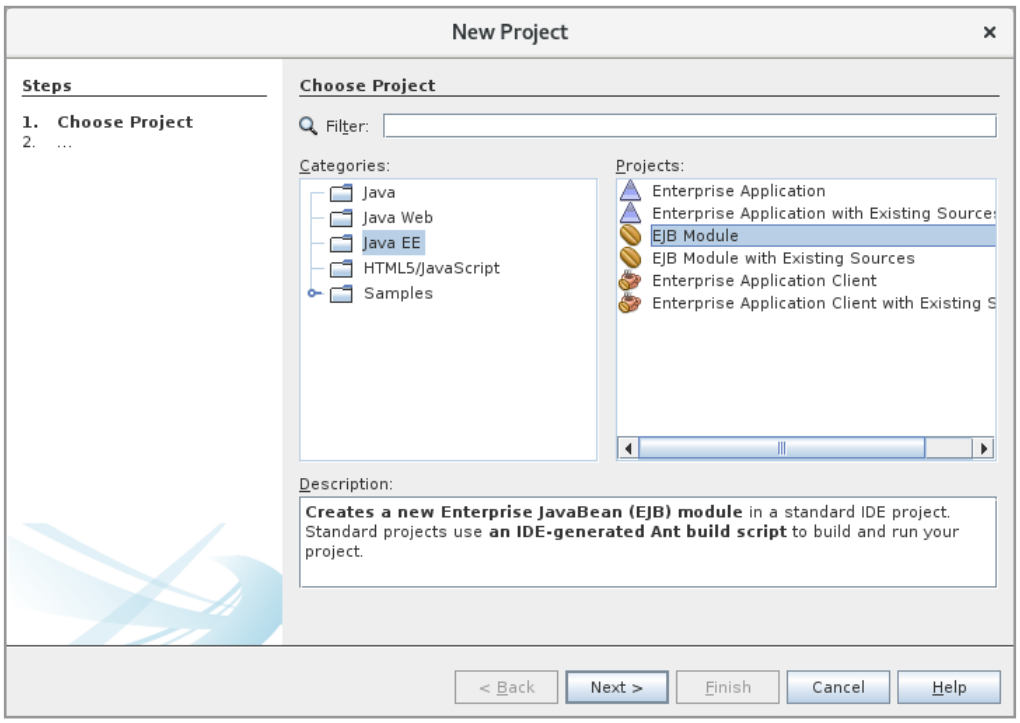
I do not want to display the code for *index.xhtml* as it does not shows anything new.

EXERCISE 2

In this exercise, write a program that performs almost the same as the example above, where you can search for Danish kings, but the user interface needs to be changed so that it also shows the rule period:



Start creating a new project that you can call *KingsEJB-ejb*, but this time it should not be an *Enterprise Application* project, but an *EJB Module* project:



Next, you should create a usual *Class Library* project, as you can call *KingsEJBRemote*. For this library you must in a package *kingsejb.ejbs* add the following class:

```
package kingsejb.ejbs;

import javax.ejb.Remote;

@Remote
public class King implements java.io.Serializable
{
 private String name;
 private int from;
 private int to;

 public King(String name, int from, int to)
 {
  this.name = name;
  this.from = from;
  this.to = to;
 }

 public String getName()
 {
  return name;
 }

 public int getFrom()
 {
  return from;
 }

 public int getTo()
 {
  return to;
 }
}
```

which represents a king. Basically, it's a common Java class, but you should note that it is decorated with *@Remote* and is *serializable*, which means that objects of this type can be sent to clients over a network.

To *KingsEJB-ejb*, you should add a singleton session bean called *KingsSession* with a remote interface (but this time no local interface) as in the previous example. The interface must be created in your class library and the content must be:

```
package kingsejb.ejbs;

import javax.ejb.Remote;
import java.util.List;

@Remote
public interface KingsSessionRemote extends java.io.Serializable
{
 List<King> getKing(int year);
 List<King> getKings(int year1, int year2);
}
```

The difference is that the two methods this time returns a list of *King* objects rather than strings. It is exactly the purpose of the exercise to show how to write a session bean, which returns custom types instead of simple java types and strings. The procedure is that the type must be defined serializable in the remote interface and decorated with *@Remote*.

You must then implement *KingsSession*, which should essentially be a copy of the class from the previous example, with the difference that the two methods will return lists of *King* objects.

Finally, you must implement the user interface, which is almost identical to the previous example, but as shown above, the table has three columns, and the *KingsBean* class must be changed slightly corresponding to the new session bean. Finally, add a converter, so years that are unknown (are 0) are shown as blank.

3.3 EJB AND JPA

In this section I will show an application that creates objects and stores them in a database when database operations are to be performed in an EJB using JPA. As an example, I will use the form for entering addresses, and compared to the last version of the previous chapter, it is only a question that the code for saving data and retrieving data has been moved from the controller to a session bean.

I have started with an Enterprise Application project with a single EJB module and a Web Application:

I have then added an entity class that represents the *Address* table in database *addresses*. The class is identical to what is shown in the previous chapter and is not showed here. Next, I have added a stateless session bean called *AddressSession* and with a local interface:

```
package addressejb.beans;

import javax.ejb.Local;
import java.util.List;

@Local
public interface AddressSessionLocal
{
 void save(Address address) throws Exception;
 List<Address> getAddresses();
}
```

Then there is the bean class itself, which implements the interface:

```
package addressejb.beans;
```

```
import java.util.*;
import javax.ejb.Stateless;
import javax.persistence.*;

@Stateless
public class AddressSession implements AddressSessionLocal
{
 @Override
 public void save(Address address) throws Exception
 {
  EntityManager em = null;
  try
  {
   em = Persistence.createEntityManagerFactory(
    "AddressEJB-ejbPU").createEntityManager();
   em.getTransaction().begin();
   em.persist(address);
   em.getTransaction().commit();
  }
  catch (Exception ex)
  {
   throw new Exception();
  }
  finally
  {
   if (em != null) em.close();
  }
 }

 @Override
 public List<Address> getAddresses()
 {
  List<Address> list;
  EntityManager em = null;
  try
  {
   em = Persistence.createEntityManagerFactory(
    "AddressEJB-ejbPU").createEntityManager();
   list = em.createNamedQuery("Address.findAll").getResultList();
  }
  catch (Exception ex)
  {
   list = new ArrayList();
  }
  finally
  {
   if (em != null) em.close();
  }
  return list;
 }
}
```

This time, an *EntityManager* is created directly without the use of injection (see next chapter). It requires that I know the persistence unit name, as shown by *persistence.xml*. In particular, note the method *save()* that uses an *EntityTransaction* and how to start this transaction. Also note the method *getAddresses()* that uses a *NamedQuery*. It was created by NetBeans in the class *Address*. The result is that *AddressSession* is a stateless session bean that encapsulates the necessary logic to update the database and execute queries. The following named bean for the user interface thus becomes similarly simpler as it delegates the work to *AddressSession*:

```
package addressejb.beans;

import javax.ejb.EJB;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.util.List;

@Named(value = "addressBean")
@SessionScoped
public class AddressBean implements java.io.Serializable
{
 @EJB
 private AddressSessionLocal addressSession;

 private Address address = new Address();
 private List<Address> addresses;

 public AddressBean() {
 }

 public Address getAddress()
 {
  return address;
 }

 public void setAddress(Address address)
 {
  this.address = address;
 }

 public List<Address> getAddresses()
 {
  return addresses = addressSession.getAddresses();
 }
```

```
 public void save()
 {
  try
  {
   addressSession.save(address);
   address = new Address();
  }
  catch (Exception ex)
  {
  }
 }
}
```

Back there are *index.xhtml* and *list.xhtml*, and the pages are essentially unchanged with the difference that the bean references have been changed. I do not want to show the two pages here.

EXERCISE 3

In this exercise you must solve the same exercise as in the previous exercise, but with the difference that the kings should be loaded from a database. The database *padata* has a table *history* that contains information about Danish kings. If the table does not exist, you can see in the book Java 6 how to create this table – the book's examples contains scripts that creates and initializes the table. To solve the exercise, you can proceed as follows:

1. Create a new Enterprise Application project called *HistoryEJB*, which should only have an EJB module. NetBeans creates the *HistoryEJB-ejb* module.
2. Create a *Class Library* project, which you can call *HistoryEJBRemote*. In addition, add the class *King* from the previous exercise.
3. Add a stateful session bean called *HistorySession* to *HistoryEJB-ejb* when it should have a remote interface that is added to *HistoryEJBRemote*. The interface should be the same as in the previous exercise (two methods that return a *List<King>*). The implementation must have an instance variable of the type *List<King>* and you must initialize it in a *@PostConstruct* method by loading data from the table *history* in the database. This you can do easy in the same way as in the previous example to create an *Entity Class from Database*.
4. Translate and deploy *HistoryEJB*.
5. Create a client, which is basically the same client as in exercise 2.
6. Test the application and that all is working.

3.4 TRANSACTIONS

One of the major advantages of using enterpris java beans is that an EJB uses transactions, which generally means that the changes a method are performed, if all are executed correctly, otherwise nothing is done corresponding to a rollback. The latter is the case if there is an exception. Immediately, it's fine, but it's not obvious how it works if a method is called while a transaction is in progress. Should the method be part of the current transaction, must the existing transaction be suspended while creating a new transaction for the new method call? Due to such questions, it is possible to configure how transactions should work. This happens with an annotation *@TransactionAttribute*:

```
@Stateless
public class Helloworld
{
 @Override
 @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
 public String hello()
 {
```

```
  return "Hello World";
 }
}
```

where the value with *TransactionAttribute* indicates how the transactions will work. There are the following options:

1. *TransactionAttributeType.MANDATORY*: The method can only be performed as part of an existing transaction. If the method is called outside of a transaction, a *TransactionRequiredException* exception is raised.
2. *TransactionAttributeType.NEVER*: If the method is called from a client, no transaction is created. If the method is called as part of a transaction, a *RemoteException* exception is raised.
3. *TransactionAttributeType.NOT_SUPPORTED*: If the method is called as part of an existing transaction, it is suspended and the method is performed outside of the transaction. If the method is called outside of a transaction, no transaction is created.
4. *TransactionAttributeType.REQUIRED*: If the method is performed as part of an existing transaction, the method will be executed as part of this transaction. If the method is called outside of an existing transaction, a new transaction will be created.
5. *TransactionAttributeType.REQUIRES_NEW*: If the method is called as part of an existing transaction, it will be suspended and a new transaction will be created. After the method is completed, the previous transaction will be resumed. If the method is called outside of an existing transaction, a new transaction is created.
6. *TransactionAttributeType.SUPPORTS*: If the method is performed as part of an existing transaction, it is performed as part of this transaction. If the method is called outside of a transaction, no new transaction is initiated.

Above is shown how to decorate a method for transactions, but you can also decorate the class and the decoration will then relate to all class's methods.

3.5 INTERCEPTION

When performing a method in a session bean, one may sometimes be interested in an action before the method is executed (when called) and immediately after the method has been completed. The typical reason is debugging, for example, you can write to in the server's log file. As an example, I have added the following class to the *AddressEJB-ejb* project:

```
package addressejb.beans;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
import java.lang.reflect.Method;

public class AddressInterceptor
{
 @AroundInvoke
 public Object logMethodCall(InvocationContext
invocationContext) throws Exception
 {
  Object interceptedObject = invocationContext.getTarget();
  Method interceptedMethod = invocationContext.getMethod();
  System.out.println("Entereding: " + interceptedObject.getClass().getName() +
   "." + interceptedMethod.getName() + "()");
  Object obj = invocationContext.proceed();
  System.out.println("Leaving: " + interceptedObject.
getClass().getName() + "." +
   interceptedMethod.getName() + "()");
  return obj;
 }
}
```

It's quite common Java class with a simple method and the first thing to notice is that the method is decorated with an annotation. The method has an *InvocationContext* parameter that provides more services. The method *getTarget()* returns the object that has performed the method, while *getMethod()* returns an object representing the method in the *interceptedObject* object that has been performed. Finally, there is the method *proceed()* which performs the method *interceptedMethod()*. On both sides of the call to *proceed()*, a *System.out.println()* is executed, which simply means writing a line in the Glassfish server's log. The result is that if a method performs *logMethodCall()*, a line is written in the log file, after which the method is executed and the log file is written again.

The question is then how to get the method executed and it happens (of course) with an annotation:

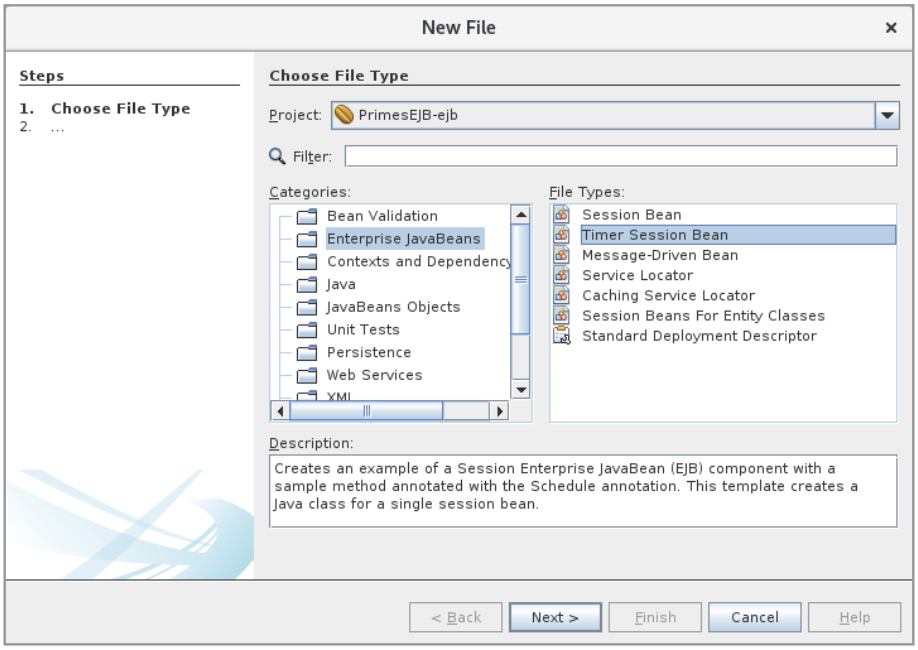
```
@Override
@Interceptors({AddressInterceptor.class})
public void save(Address address) throws Exception
{
 EntityManager em = null;
 try
 {
  em = Persistence.createEntityManagerFactory("AddressEJB-ejbPU").
   createEntityManager();
  em.getTransaction().begin();
  em.persist(address);
  em.getTransaction().commit();
 }
 catch (Exception ex)
 {
  throw new Exception();
 }
 finally
 {
  if (em != null) em.close();
 }
}
```

That is, writing in the log file before creating an address and again after the database is updated.

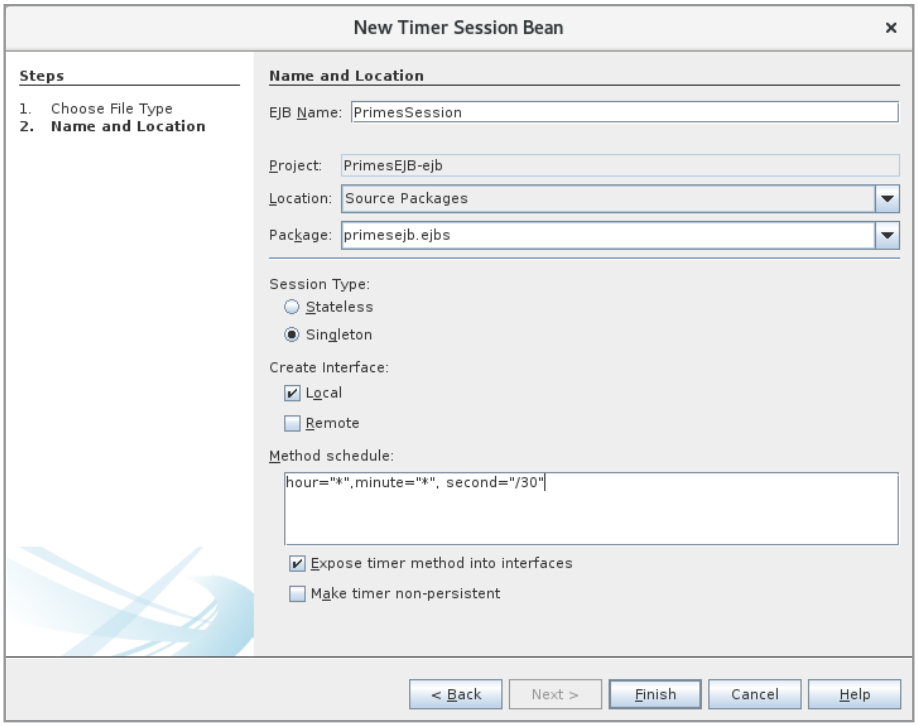
3.6 A TIMER SERVICE

There is a special variant of a session bean that implements a timer. You can therefore have the actual bean to perform an operation at certain intervals, for example, you can write in the log file. In this case, I will write a session bean that determines the next prime every

half second. I start with an *Enterprise Application* project named *PrimesEJB* with an EJB Module and a Web Application. For the EJB Module, I have added an EJB, but this time it's a *Timer Session Bean*:



You will then get the following window, where you should as usual must enter a name (here *PrimesSession*) and a package (*primesejb.ejbs*). I have also selected that it must be a *Singleton* and that a *Local* interface should be created:



Finally, I have set the time interval for when the timer has to tick. In this case, the pattern means that the timer must tick every half-minute. After clicking *Finish*, NetBeans creates the following class:

```
package primesejb.ejbs;

import javax.ejb.Schedule;
import javax.ejb.Singleton;

@Singleton
public class PrimesSession implements PrimesSessionLocal
{
 @Schedule(hour = "*", minute = "*", second = "*/30")
 public void myTimer()
 {
 }
}
```

as well as a defining interface. As can be seen, it is a usual session bean with the only difference that a timer method has been created, with a *@Schedule* annotation defining that it is a timer function and how often it should tick. The finished interface is:

```
package primesejb.ejbs;

import javax.ejb.Local;
import java.math.BigInteger;

@Local
public interface PrimesSessionLocal
{
 public void myTimer();
 public void reset();
 public BigInteger getPrime();
 public int getIndex();
}
```

where in addition to the timer, three methods are defined. The class *PrimesSession* will implement the interface:

```
package primesejb.ejbs;

import java.math.BigInteger;
import javax.ejb.Schedule;
import javax.ejb.Singleton;

@Singleton
public class PrimesSession implements PrimesSessionLocal
{
 private BigInteger two = new BigInteger("2");
 private BigInteger prime = new BigInteger("2");
 private int index = 0;

 @Schedule(hour = "*", minute = "*", second = "*/30")
 public void myTimer()
 {
  if (prime.equals(two)) prime = new BigInteger("3");
  else
  {
   prime = prime.add(two);
   while (!prime.isProbablePrime(50)) prime = prime.add(two);
  }
  ++index;
 }

 public void reset()
 {
  prime = two;
  index = 0;
 }
```

```
 public BigInteger getPrime()
 {
  return prime;
 }

 public int getIndex()
 {
  return index;
 }
}
```

You should note that the class has three instance variables, and since it is a singleton session bean, these variables will retain their values from the class *PrimesSession* is initialized for the first time until the Glassfish server is stopped. That is, starting the application multiple times, the state of the session bean will be preserved. The most important thing about the class is, of course, the timer method, which modifies the value of the variable *prime* as it represents the next prime. The timer method also modifies the variable *index*.

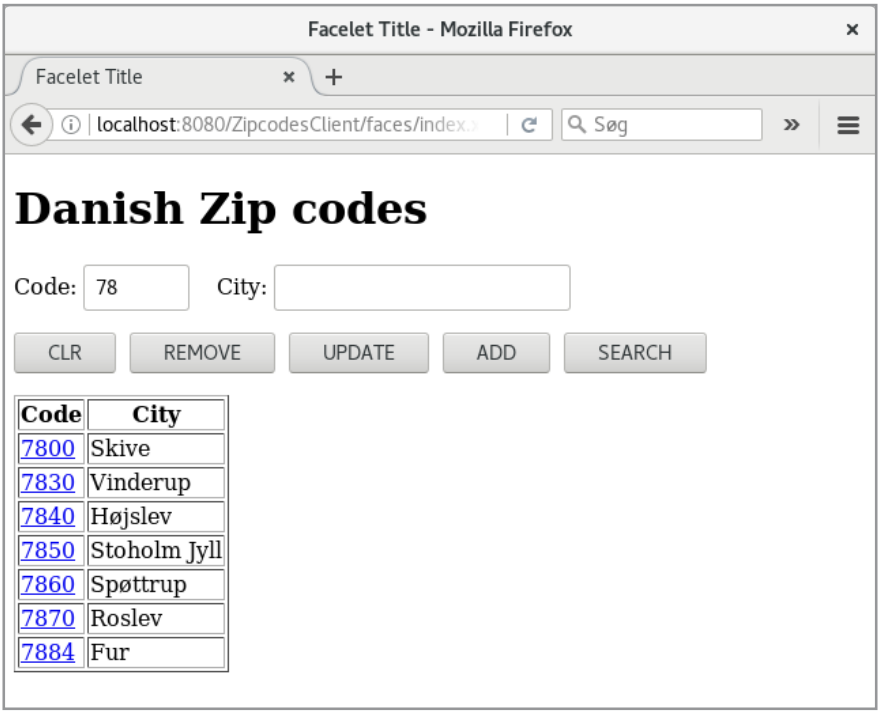
Then the EJB module is complete. I have added a named bean to the web application:

```
package primesejb.beans;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import java.math.BigInteger;
import javax.ejb.EJB;
import primesejb.ejbs.PrimesSessionLocal;

@Named(value = "primesBean")
@RequestScoped
public class PrimesBean
{
 @EJB
 private PrimesSessionLocal primesSession;

 public PrimesBean()
 {
 }

 public BigInteger getPrime()
 {
  return primesSession.getPrime();
 }
```

```
 public int getIndex()
 {
  return primesSession.getIndex();
 }

 public void reset()
 {
  primesSession.reset();
 }
}
```

It's a very simple bean and there's not much to notice, but you need to be aware that it uses my session bean. The JSF page is similarly simple and opens it in the browser, the result is as shown below, where the link is used to update the user interface:

3.7 CRUD WITH ONE TABLE



In database contexts, CRUD stands for *Create, Read, Update* and *Delete* operations, and in this section I will show you how to write a program that performs these operations on a single database table using an EJB. The application must use an EJB with a remote interface. As an example, I will use the table *zipcode* from the database *padata*, and the application should open a window as shown above, where 78 entered for zip code has been searched. The meaning of the buttons should be self explanatory and clicking on a link in the table will insert the zip code into the top two fields where they can be edited. In the following, I will explain how the program is written and thus the process.

I have started creating an *EJB Module* project, which I have called *ZipcodesEJB*. Then I have created a usual *Class Library* project, which I have called *ZipcodesRemote*. For this project, I have added a class *ZipcodeRemote*, which should represent the objects to be transmitted between the web application and the session bean object (I have not shown *get* and *set* methods):

```
package zipcodes.beans;

import javax.ejb.Remote;

@Remote
public class ZipcodeRemote implements java.io.Serializable
{
 private String code;
 private String city;

 public ZipcodeRemote()
 {
 }
```

Then I have to the project *ZipcodesEJB* added a remote stateless session bean named *ZipcodesSession*, which adds a remote interfaces to the above class library:

```
package zipcodes.beans;

import javax.ejb.Remote;
import java.util.List;

@Remote
public interface ZipcodesSessionRemote
{
 public boolean create(ZipcodeRemote obj);
 public boolean update(ZipcodeRemote obj);
 public boolean remove(String code);
 public List<ZipcodeRemote> search(String code, String City);
}
```

The biggest work is to implement the class *ZipcodesSession*. In the *zipcodes.models* package I have added an entity class *Zipcode* for the database table. This class is unchanged from the code generated by NetBeans. I would like to add a class for database operations, but a JPA controller class can not be directly used from an EJB, so I added my own (which is nothing but a modified and scaled-down version of the controller class created by NetBeans – see also the next example), which I have called *ZipcodeDAO*:

```
package zipcodes.models;

import java.util.*;
import javax.persistence.*;

public class ZipcodeDAO
{
 public EntityManager getEntityManager()
 {
  return Persistence.
    createEntityManagerFactory("ZipcodesEJBPU").createEntityManager();
 }

 public void create(Zipcode zipcode) throws Exception
 {
  EntityManager em = null;
  try
  {
   em = getEntityManager();
   em.getTransaction().begin();
   em.persist(zipcode);
   em.getTransaction().commit();
  }
```

```
 catch (Exception ex)
 {
  if (em != null) em.getTransaction().rollback();
  throw ex;
 }
 finally
 {
  if (em != null) em.close();
 }
}

public void edit(Zipcode zipcode) throws Exception
{
 EntityManager em = null;
 try
 {
  em = getEntityManager();
  em.getTransaction().begin();
  em.merge(zipcode);
  em.getTransaction().commit();
 }
 catch (Exception ex)
 {
  if (em != null) em.getTransaction().rollback();
  throw ex;
 }
 finally
 {
  if (em != null) em.close();
 }
}

public void destroy(String code) throws Exception
{
 EntityManager em = null;
 try
 {
  em = getEntityManager();
  em.getTransaction().begin();
  Zipcode zipcode;
  try
  {
   zipcode = em.getReference(Zipcode.class, code);
   zipcode.getCode();
  }
  catch (Exception e)
  {
   throw new Exception("The zipcode with id " + code + " no longer exists");
  }
```

```
  em.remove(zipcode);
  em.getTransaction().commit();
 }
 catch (Exception ex)
 {
  if (em != null) em.getTransaction().rollback();
  throw ex;
 }
 finally
 {
  if (em != null) em.close();
 }
}

public List<Zipcode> getZipcodes(String code, String city)
{
 if (code == null) code = "";
 if (city == null) city = "";
 EntityManager em = null;
 try
 {
  em = getEntityManager();
  Query cq = em.createQuery(
   "SELECT z FROM Zipcode z WHERE z.code like :code AND z.city like :city");
  cq.setParameter("code", code + "%");
  cq.setParameter("city", "%" + city + "%");
  return cq.getResultList();
 }
 catch (Exception ex)
 {
  return new ArrayList();
 }
 finally
 {
  if (em != null) em.close();
 }
}
```

The class is very similar to the DAO (*Data Access Object*) classes that NetBeans creates, but you should note how to create an *EntityManager* object and how to define transactions associated with *Create*, *Update* and *Delete*. Also note the last method, which is the search method. Here, JPQL is used instead of the Criteria API. It's a matter of taste, if you do one or the other, but if you know SQL, it's easiest to use JPQL.

With these two classes in place, you can write the class *ZipcodesSession*, where the code is actually quite simple as all the work is delegated to the DAO class:

```
package zipcodes.beans;

import java.util.*;
import javax.ejb.Stateless;

import zipcodes.models.*;

@Stateless
public class ZipcodesSession implements ZipcodesSessionRemote
{
 @Override
 public boolean create(ZipcodeRemote obj)
 {
  try
  {
   (new ZipcodeDAO()).create(new Zipcode(obj.getCode(), obj.getCity()));
   return true;
  }
  catch (Exception ex)
  {
   return false;
  }
 }
```

```
 @Override
 public boolean update(ZipcodeRemote obj)
 {
  try
  {
   (new ZipcodeDAO()).edit(new Zipcode(obj.getCode(), obj.getCity()));
   return true;
  }
  catch (Exception ex)
  {
   return false;
  }
 }

 @Override
 public boolean remove(String code)
 {
  try
  {
   (new ZipcodeDAO()).destroy(code);
   return true;
  }
  catch (Exception ex)
  {
   return false;
  }
 }

 @Override
 public List<ZipcodeRemote> search(String code, String City)
 {
  return remoteList((new ZipcodeDAO()).getZipcodes(code, City));
 }

 private List<ZipcodeRemote> remoteList(List<Zipcode> zipcodes)
 {
  List<ZipcodeRemote> list = new ArrayList();
  for (Zipcode zipcode : zipcodes)
   list.add(new ZipcodeRemote(zipcode.getCode(), zipcode.getCity()));
  return list;
 }
}
```

Note that the methods contains conversions of objects of type *ZipcodeRemote* to and from objects of the entity type *Zipcode*. It is hardly necessary and the problem could be solved by defining the entity type in the class library (see the next example).

Then the EJB module is ready and ready to be hosted on the server. Back there is the web application where I have created a Web Application project named *ZipcodesClient*. The application consists only of *index.xhtml* and a single named bean, and I do not want to display the code for the application as there is nothing new.

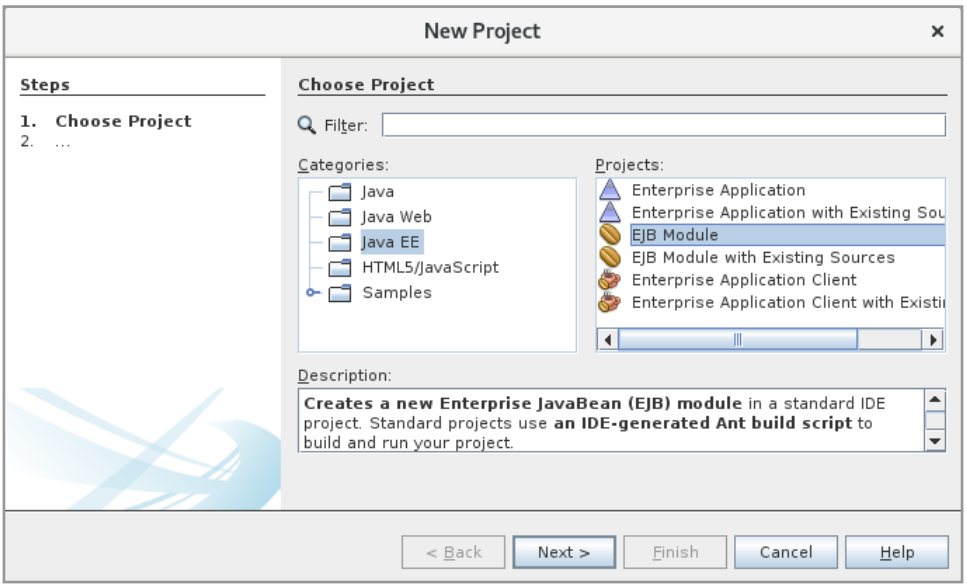
3.8 CRUD WITH MORE TABLES

As the last example of Enterprise Java Beans, I will review the *PaBooks* program from the previous chapter. It must be completely the same program with only one change in which the entire logic of database operations must be moved to an EJB, which uses JPA to maintain the database. There will be three projects:

1. *BooksEJB*, which is an Enterprise Java Bean hosted on the server, and which can maintain the *Library* database.
2. *BooksRemote*, which is a class library that defines a remote interface to the above EJB.
3. *BooksClient*, which is the web application, which uses *BooksEJB*.

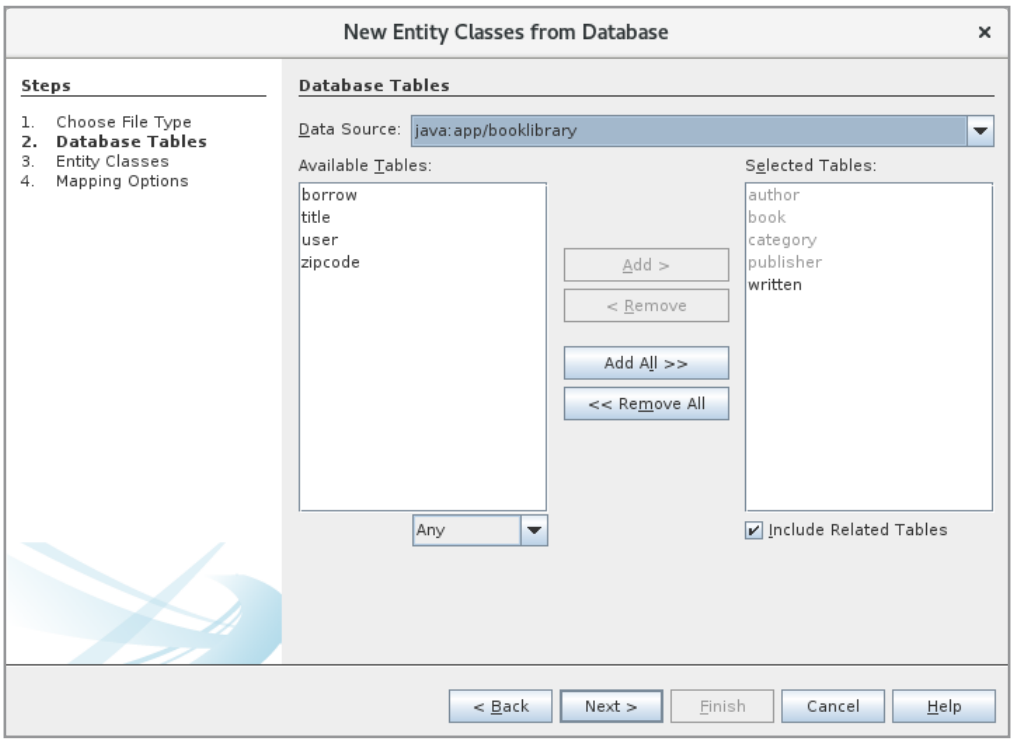
In the following I will describe the process and the main challenges regarding the application development.

I have started creating an *EJB Module* project:

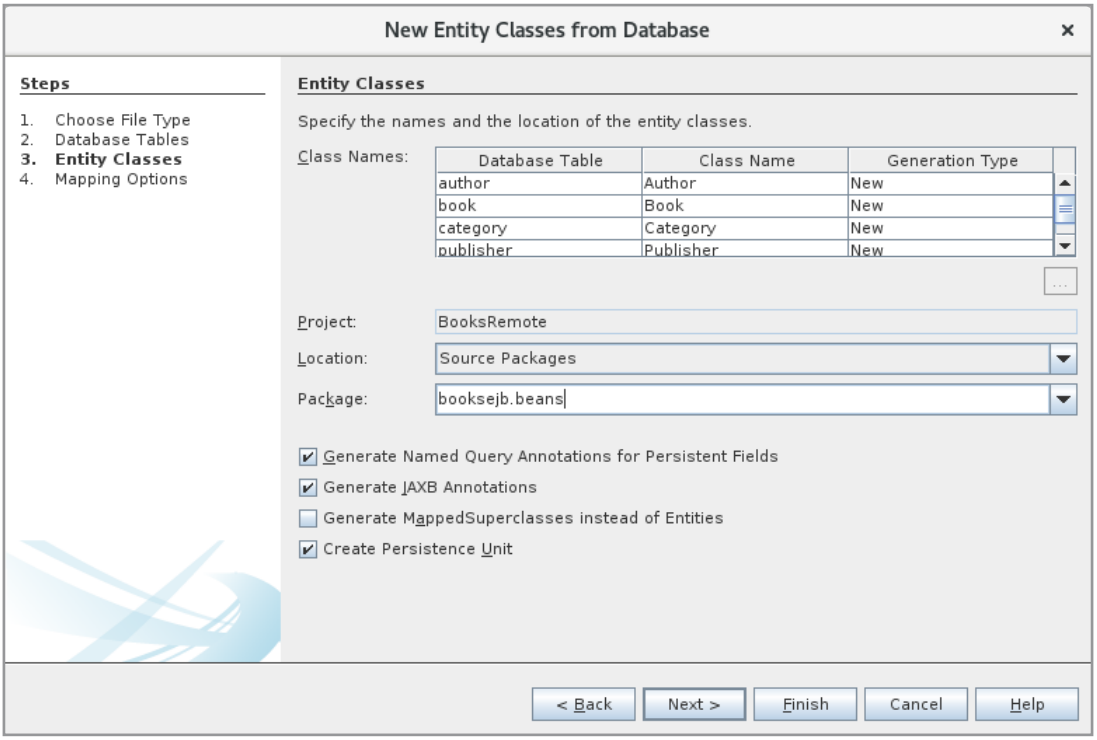


I have called the project *BooksEJB*.

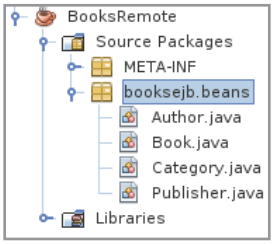
As a next step, I've created a classic class library project called *BooksRemote*. For this library I have selected *Entity Classes from Database* and selected a connection to the *library* database:



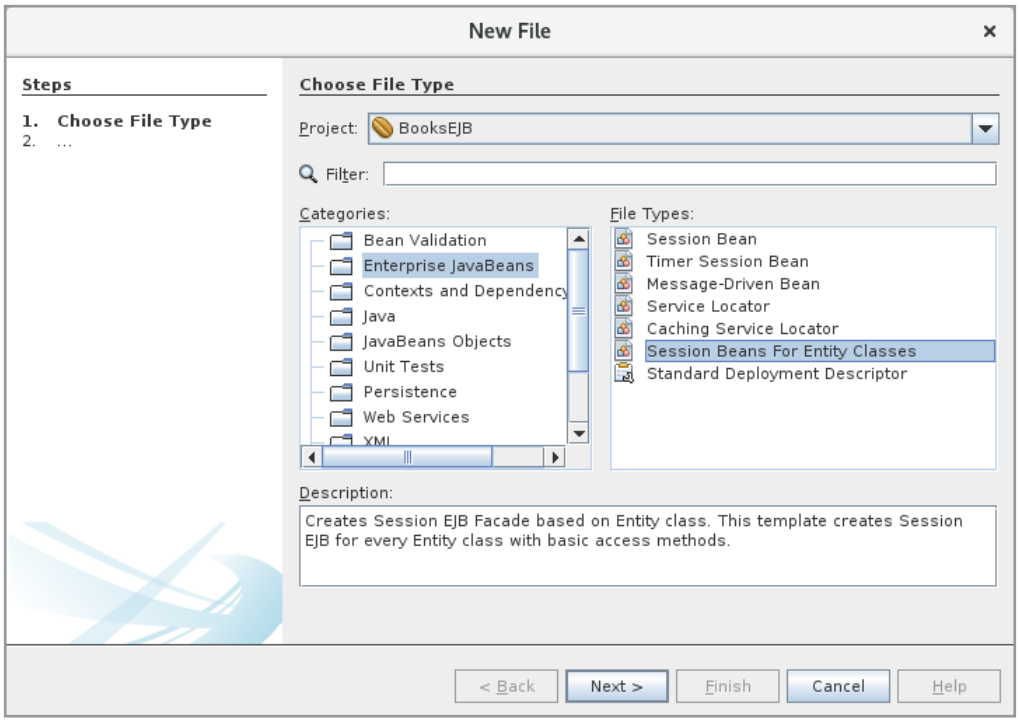
I have then chosen the 5 tables that I will apply and in the following window I have entered the name of a package:



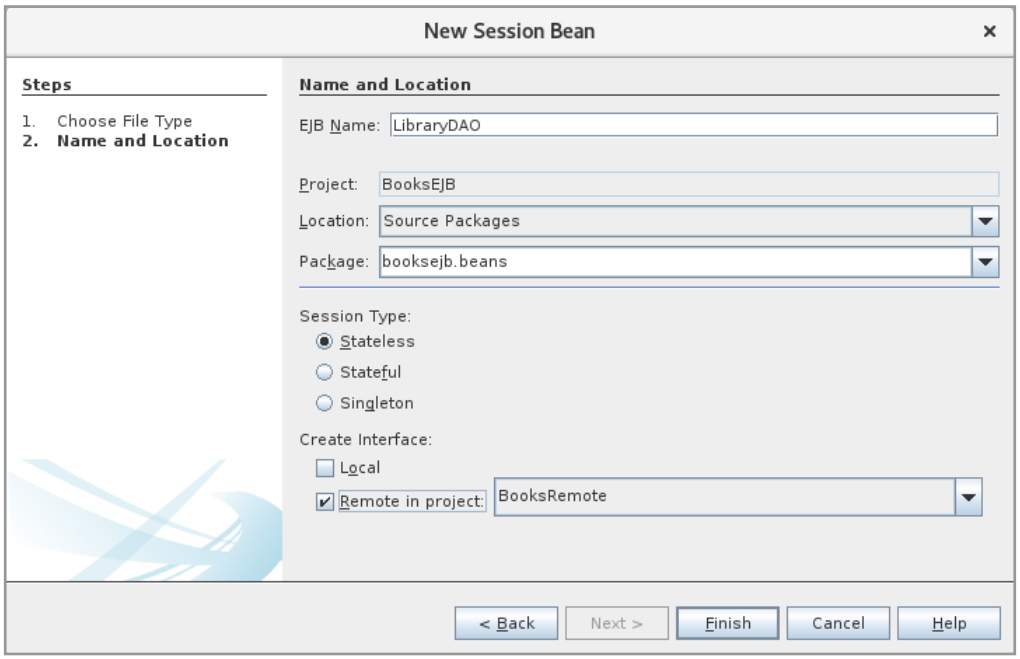
When I then click Next and Finish, there are created as before 4 entity classes:



You should note that the four classes are located in the class library so that they can be used by the client program. The four classes should not be changed in relation to the code created by NetBeans, except for I, for *Author*, *Publisher* and *Category*, have changed *toString()*.

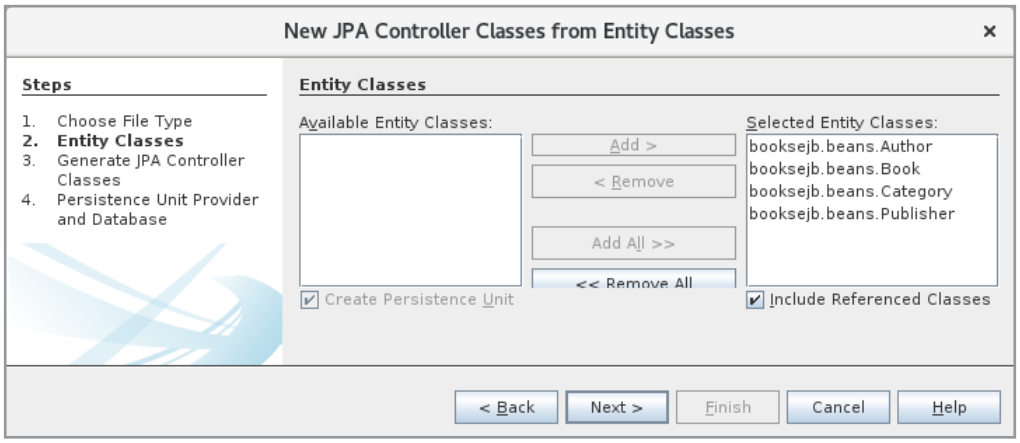


As a next step, I have added to the project *BooksEJB* a stateless session bean:



I have defined that a remote interface has to be added to my class library, and the result is that NetBeans creates an interface named *LibraryDAORemote*.

In the *BooksEJB* project I have created a package called *booksejb.daos*. For this package I have added a *JPA Controller Classes from Entity Classes*:



These 4 classes are all modified a bit, but it is solely a question that the constructor has been removed that the *getEntityManager()* method has been changed and the transactions are defined differently. The code for these classes fills a lot and I do not want to show them here, but below is the start of the class *CategoryDAO* and the method *create()*:

```
public class CategoryDAO
{
 public EntityManager getEntityManager()
 {
  return Persistence.createEntityManagerFactory("BooksRemotePU").
   createEntityManager();
 }

 public void create(Category category) throws Exception
 {
  if (category.getBookCollection() == null)
   category.setBookCollection(new ArrayList<Book>());
  EntityManager em = null;
  try
  {
   em = getEntityManager();
   em.getTransaction().begin();
   Collection<Book> attachedBookCollection = new ArrayList<Book>();
   for (Book bookCollectionBookToAttach : category.getBookCollection())
   {
    bookCollectionBookToAttach =
     em.getReference(bookCollectionBookToAttach.getClass(),
     bookCollectionBookToAttach.getId());
    attachedBookCollection.add(bookCollectionBookToAttach);
   }
   category.setBookCollection(attachedBookCollection);
   em.persist(category);
   for (Book bookCollectionBook : category.getBookCollection())
   {
    Category oldCatidOfBookCollectionBook = bookCollectionBook.getCatid();
    bookCollectionBook.setCatid(category);
    bookCollectionBook = em.merge(bookCollectionBook);
    if (oldCatidOfBookCollectionBook != null)
    {
     oldCatidOfBookCollectionBook.getBookCollection().
      remove(bookCollectionBook);
     oldCatidOfBookCollectionBook = em.merge(oldCatidOfBookCollectionBook);
    }
   }
   em.getTransaction().commit();
  }
  catch (Exception ex)
  {
   if (em != null) em.getTransaction().rollback();
   throw ex;
  }
  finally
  {
   if (em != null) em.close();
  }
 }
```

In addition, I have changed the classes so that they alone raises an *Exception* and deleted the other exception types. Of course, there is no special reason for and is something of an attitude question, but I feel that it results in a code that is slightly easier to read.

Finally, the *BookDAO* class has added a method that determines *Book* objects for the search criteria used, but it is the same method that is written in the *PaBooks* program.

Note that until this place I have not written quite a lot of code, but NetBeans has essentially created it all. All that is missing is to define the *LibraryDAORemote* interface and implement *LibraryDAO*. The interface is defined as follows:

```
package booksejb.beans;

import javax.ejb.Remote;
import java.util.List;

@Remote
public interface LibraryDAORemote
{
 public boolean addCategory(Category category);
 public boolean modCategory(Category category);
 public boolean delCategory(Category category);
 public List<Category> getCategories();
 public boolean addPublisher(Publisher publisher);
 public boolean modPublisher(Publisher publisher);
 public boolean delPublisher(Publisher publisher);
 public List<Publisher> getPublishers();
 public boolean addAuthor(Author author);
 public boolean modAuthor(Author author);
 public boolean delAuthor(Author author);
 public List<Author> getAuthors();
 public boolean addBook(Book book);
 public boolean modBook(Book book);
 public boolean delBook(Book book);
 public List<Book> getBooks(String title, String pubname, String catname);
}
```

The names of the individual methods should explain what they should be used for. Since there are 12 methods (4 for each entity), the *LibraryDAO's* code is a part, but since the implementation is essentially the same for each entity, I will show only the code for the entity *Category*:

```
package booksejb.beans;

import javax.ejb.Stateless;
import java.util.List;

import booksejb.daos.*;

@Stateless
public class LibraryDAO implements LibraryDAORemote
{
 @Override
 public boolean addCategory(Category category)
 {
  try
  {
   CategoryDAO dao = new CategoryDAO();
   dao.create(category);
   return true;
  }
  catch (Exception ex)
  {
   return false;
  }
 }
```

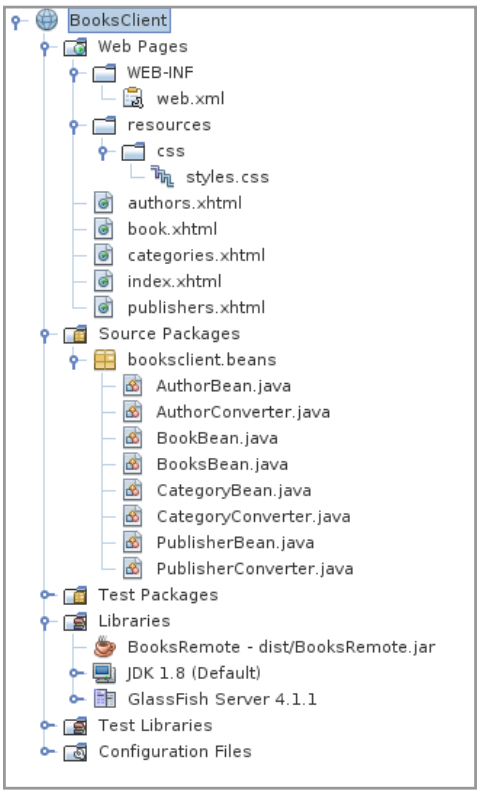
```
 @Override
 public boolean modCategory(Category category)
 {
  try
  {
   CategoryDAO dao = new CategoryDAO();
   dao.edit(category);
   return true;
  }
  catch (Exception ex)
  {
   return false;
  }
 }

 @Override
 public boolean delCategory(Category category)
 {
  try
  {
   CategoryDAO dao = new CategoryDAO();
   dao.destroy(category.getId());
   return true;
  }
  catch (Exception ex)
  {
   return false;
  }
 }

 @Override
 public List<Category> getCategories()
 {
  List<Category> list = (new CategoryDAO()).findCategoryEntities();
  for (Category c : list) c.getBookCollection().size();
  return list;
 }
}
```

The code is simple, as all work is performed by the DAO classes, and in principle, it is about creating an DAO object and calling the current DAO's methods. However, there is a single problem that requires a special solution. JPA uses what is called LAZY update. For example, if you take the entity class *Category*, it has a collection *bookCollection* that contains a *Book* object for all the books that are categorized under that category. These objects must thus be initialized when you read the database, and since these objects are far from always used, JPA use a technique where the objects are first created (by reading in the database) when referring to them. Of course, the reason is performance, and you should keep in mind that

all categories have such a collection, and the same goes for publishers, authors and books. It is this technique, that is called LAZY update, but there is a problem when objects are to be serialized of a remote client, as they can risk the objects being serialized before the relevant collections are updated. There are various suggestions on how to solve this problem, and one of them is – and probably the simplest, but perhaps not the best – that before the objects are serialized, the method performs the *size()* method of the collections in question (resulting in the objects being initialized by reading in the database). This is the purpose of the loop that occurs in the method *getCategories()*.

After the *LibraryDAO* class is implemented, *BooksEJB* is complete and can be translated and hosted on the server.

Back there is the client program, which can be copied essentially from *PaBooks*. I have created a Web Application project named *BooksClient*. The project should have a reference to *BooksRemote*, but otherwise there should be the following files that can be copied from PaBooks:



However, the five named bean classes must be changed a bit, and as an example, I have shown *CategoryBean*, which is a named bean for *categories.xhtml*:

```
package booksclient.beans;

import booksejb.beans.LibraryDAORemote;
import javax.ejb.EJB;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.util.*;

import booksejb.beans.*;

@Named(value = "categoryBean")
@SessionScoped
public class CategoryBean implements java.io.Serializable
{
 @EJB
 private LibraryDAORemote libraryDAO;
 private Category category = new Category();
 private String error = "";

 public CategoryBean()
 {
 }
```

```
public int getId()
{
 return category.getId() == null ? 0 : category.getId();
}

public String getName()
{
 return category.getName();
}

public void setName(String name)
{
 this.category.setName(name);
}

public String getError()
{
 return error;
}

public Collection<Category> getCategories()
{
 return libraryDAO.getCategories();
}

public void select(Category cat)
{
 category = cat;
}

public void clear()
{
 error = "";
 category = new Category();
}

public void add()
{
 error = "";
 if (libraryDAO.addCategory(category)) category = new Category();
 else error = "Category could not be created!";
}

public void update()
{
 error = "";
 if (libraryDAO.modCategory(category)) category = new Category();
```

```
  else error = "Category could not be updated!";
 }

 public void remove()
 {
  error = "";
  if (libraryDAO.delCategory(category)) category = new Category();
  else error = "Category could not be deleted!";
 }
}
```

After *BooksClient* is translated and hosted on the server, the program can be tested and the result should be the same as the *PaBooks* application.

At the beginning of this chapter, I noticed that it may be difficult to develop and test Enterprise Java Beans. As for writing the code, it is not very different from other Java code, but it may be problematic to get a project deployed to the server and in case of trouble to find out what the reason is. Here you should note the log file:

```
/usr/local/glassfish-4.1.1/glassfish/domains/domian1/logs
```

or where the Glassfish server should now be installed. The server writes in this log file every time you deploy a project and especially if not succeed and in that context also the cause of the problem. This log file is a very important source of debugging (can also be accessed from NetBeans).

PROBLEM 2

You must solve the same task as in problem 1, but with the difference that the program should be written in the same way as shown in the previous example, where the database operations are moved to an EJB, which maintains the database using JPA. You must thus write three projects:

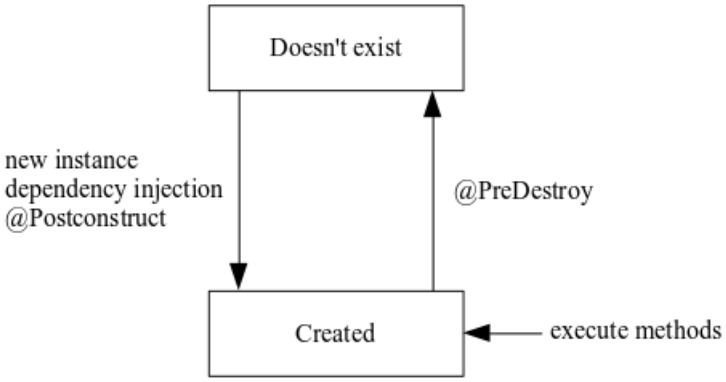
1. *WorldRemote*, which is a remote interface
2. *WorldEJB*, which is a EJB Module projct
3. *WorldClient*, which is a web applicationen

4 CDI

CDI stands for *Context and Dependency Injection* and describes how the application server creates and remove objects, and especially what you can and should write in the code. For example, if you consider a JSF page with a backing bean, which is just a common Java class, then you do not need to do anything to create an object of the class and how long the object lives is determined by its scope with an annotation.

A JavaBean is just a class that is written from a specific and simple pattern, and the programmer creates an object (a bean) using the *new* operator. After that, it is the garbage collector who is responsible for removing the object when there are no longer any references to it. In contrast, a managed bean is a bean, where it is the container in the form of the application server, which is responsible for creating an object and removing the object again when the program goes out of the the object's scope. Such a bean is also called a CDI object. You can not create a managed bean with *new*. CDI objects are instead created using a technique called *injection*, which means that it is the container that takes care of it. When defining a bean with *injection*, the container will examine whether the bean is created and if not, it will instantiate a new object and perform injection on all beans to which the current bean depends and finally, a method decorated with *@PostConstruct* is performed.

Then you can use the object's methods. When the bean has to be removed, the container will first perform a method decorated with *@PreDestroy*, after which it is the container that remove the object. It can be illustrated by the following figure:



CDI objects have a scope that is called *context*, and it can basically be

- request
- session
- application
- conversation

As an example, below is shown a simple managed bean:

```
package cdiprogram.beans;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
@Named(value = "person")
@SessionScoped
public class Person implements Serializable
{
 private String firstName;
 private String lastName;

 public Person()
 {
 }

 public String getFirstName()
 {
  return firstName;
 }
```

```
 public void setFirstName(String firstName)
 {
  this.firstName = firstName;
 }

 public String getLastName()
 {
  return lastName;
 }

 public void setLastName(String lastName)
 {
  this.lastName = lastName;
 }
}
```

That it's a managed bean you can see from *@Named*, where you can see that NetBeans has assigned the name for the class written with lower case. An annotation is generally an interface, and in this case it is an interface with a single attribute *value* (whose default value is blank, meaning that the container selects the class name *person*). In addition, you can see that an object's context is *session*, and *SessionScoped* is again an interface. The two interfaces *Named* and *SessionScoped* are thus information to the container, which tells how the container should maintain objects of the class *Person*.

With the above named bean available you can write the following JSF page:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  <h:form>
   <h:panelGrid columns="2">
    <h:outputLabel for="firstName" value="First Name"/>
    <h:inputText id="firstName" value="#{person.firstName}"/>
    <h:outputLabel for="lastName" value="Last Name"/>
    <h:inputText id="lastName" value="#{person.lastName}"/>
    <h:commandButton value="OK" />
   <h:panelGroup/>
   </h:panelGrid>
  </h:form>
 </h:body>
</html>
```

The interesting thing is to open the page, the Glassfish server will inject a *Person* object with context *session*, without the programmer being – or can – doing something to create the object.

In the previous examples I have often had to use a named bean from another named bean, which has been done by means of statements of the form

```
ELContext elc = FacesContext.getCurrentInstance().getELContext();
SomeBean someBean = (SomeBean)FacesContext.
getCurrentInstance().getApplication().
 getELResolver().getValue(elc, null, "person");
```

There's nothing wrong with it, but it's hard to write and impossible to remember. One can, however, facilitate the syntax by the use of CDI. Consider the following named bean, which represents a student with a name and a *subject*:

```
package cdiprogram.beans;

import javax.inject.Named;
import javax.inject.Inject;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
```

```
@Named(value = "student")
@SessionScoped
public class Student implements Serializable
{
 @Inject
 private Person person;
 private String subject;

 public Student()
 {
 }

 public Person getPerson()
 {
  return person;
 }

 public void setPerson(Person person)
 {
  this. person = person;
 }

 public String getSubject()
 {
  return subject;
 }

 public void setSubject(String subject)
 {
  this.subject = subject;
 }
}
```

The name is represented by a *person* variable of the type *Person*. However, you must note that there nowhere are created an object of the type *Person* (the property seams not to be initialized), but instead, a person is decorated with *@Inject*. It stated that a *Student*, which is a managed bean, is dependent on *Person* (another managed bean), and when the container creates a *Student* object with the help of injection, it will also look for whether a *Person* object has to be created and if so, do it. Below is shown how to write a JSF page that has *Student* as a backing bean:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
```

```
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  <h:form>
   <h:panelGrid columns="2">
    <h:outputLabel for="firstName" value="First Name"/>
    <h:inputText id="firstName" value="#{student.person.firstName}"/>
    <h:outputLabel for="lastName" value="Last Name"/>
    <h:inputText id="lastName" value="#{student.person.lastName}"/>
    <h:outputLabel for="subject" value="Subject"/>
    <h:inputText id="subject" value="#{student.subject}"/>
    <h:commandButton value="OK" />
   <h:panelGroup/>
   </h:panelGrid>
  </h:form>
 </h:body>
</html>
```

4.1 QUALIFIERS

If you consider the bean *Student*, it has a dependence to the bean *Person*. I will now define a derivative class, which I have called *Man*, but first I will made the class *Person* abstract and expanded with an abstract method that will return a person's gender:

```
@Named(value = "person")
@SessionScoped
public abstract class Person implements Serializable
{
 …

 public abstract String getGender();
}
```

I can then write the following bean:

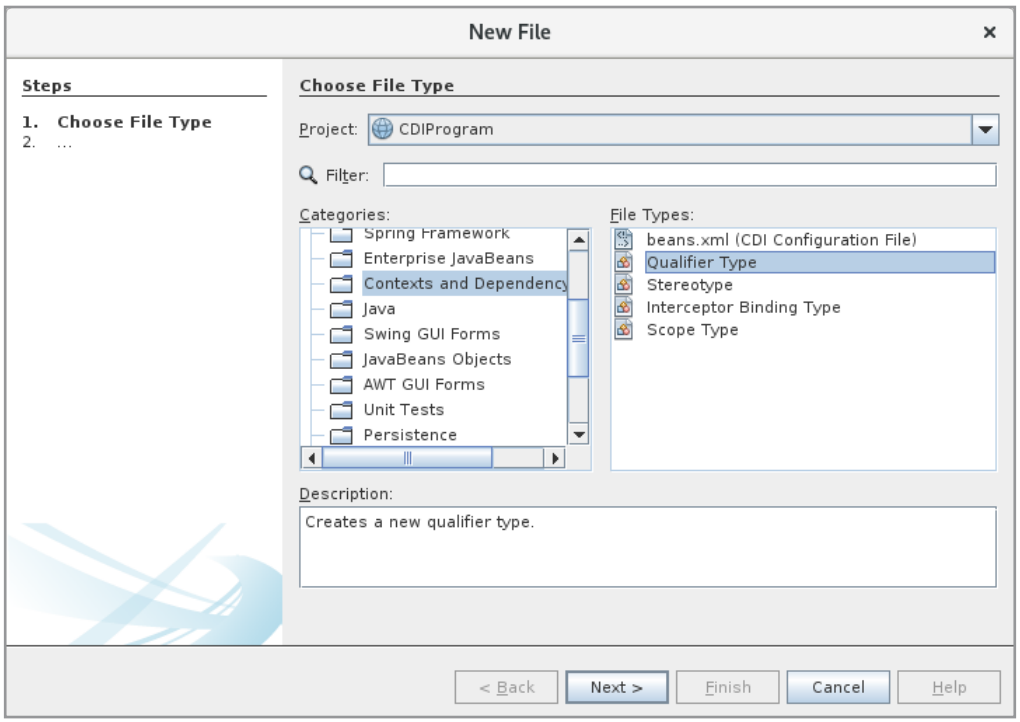
```
package cdiprogram.beans;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;

@Named(value = "man")
@SessionScoped
@Male
```

```
public class Man extends Person implements Serializable
{
 public Man()
 {
 }

 public String getGender()
 {
  return "A man";
 }
}
```

If you disregard the *@Male* annotation, there is nothing new in this class. It's a managed bean that inherits a managed bean, and a bean is just a class. If you see the *Student* class from the container, it defines a variable *person* whose type is *Person*, and the container must then create an actual object, such as a *Man* object. Since the class *Person* could have more derived classes, one should be able to tell the container what it is for an object to be instantiated and it happens with a *qualifier*. *Male* is a qualifier and is an interface that can be defined as follows:

```
package cdiprogram.beans;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Male
{
}
```

In the class *Man*, you tell with *@Male* that the class is identified with the qualifier *Male*. The *CDIProgram* project also defines a derived class *Woman*, identified with a qualifier *Female*. In principle, it's simple to write a qualifier, but the syntax can be hard to remember, and NetBeans therefore has a *File Type* for the purpose:



If you then want the Student class to create a *Woman* object, the syntax is as follows:

```
public class Student implements Serializable
{
 @Inject
 @Female
 private Person person;
```

Looking at the above solution, it is simple enough to define the specific type the container has to use in conjunction with injection, but if you think of a situation where the class *Person* has many sub classes, many qualifications must be written. Instead, you can define a parameter. Consider first the following *enum*:

```
package cdiprogram.beans;
public enum People { MALE, FEMALE }
```

With this type available, I can create another qualifier:

```
package cdiprogram.beans;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Gender
{
```

This qualifier now has an attribute called *value*. Note that a qualifier may well have more attributes. You can then decorate the classes *Man* (and *Woman*) as follows:

```
@Named(value = "man")
@SessionScoped
@Gender(value = People.MALE)
public class Man extends Person implements Serializable
{
```

and for injection you can write:

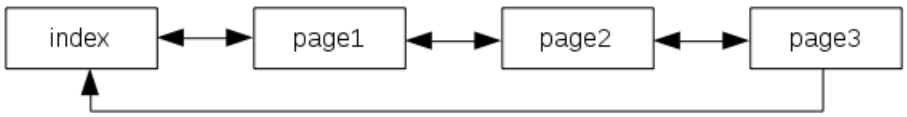
```
@Named(value = "student")
@SessionScoped
public class Student implements Serializable
{
 @Inject
 @Gender(value = People.FEMALE)
 private Person person;
 private String subject;
```

4.2 CONTEXTS

When a managed bean is needed either by injection or because it is referenced from a JSF page, CDI will try to find an instance in the scope of the particular bean. If successful, this object is used, but otherwise an object belonging to the current scope will be created. Often, one object (a managed bean) must "live" across multiple pages, and you can not use *RequestScoped* as it relates only to the current page. The object is created each time a page request is made. So far (both in this book and the two previous books) I have solved this

problem by defining the beans as *SessionScoped*. It works, but is also the easy solution, and there is also a disadvantage to *SessionScoped* beans. Firstly, these beans live throughout the session and even for a period after the session is completed. This means that session objects fill unnecessarily in the machine's memory. An even bigger problem, however, is that since *SessionScoped* beans (and, for that matter, all session objects) live longer than is needed for them, there is a risk that they will be used after the logically no longer exist. The latter is actually something that can make it very difficult to correct errors in a web application.

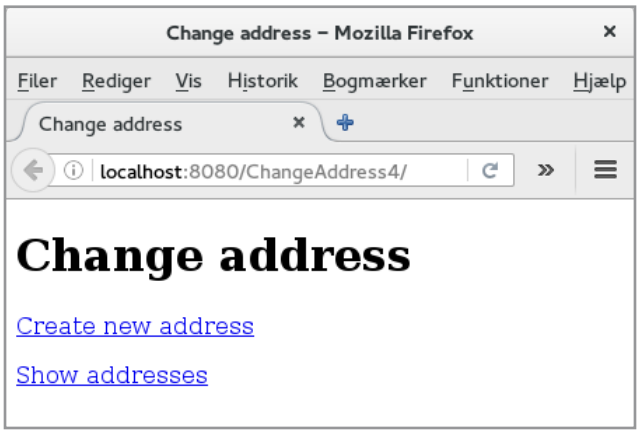
There is a context called *conversation*, and where a bean can live over several pages, but not through an entire session, and which precisely aims to solve the challenges that the session context poses. The price is that it is the programmer who has to tell the container when a conversation starts and when it ends. I want to show how with the *ChangeAddress* example, where the difference should be that the form for entering the address is divided into three pages – without any reason, in addition to showing how *ConversationScoped* works. I start with a project called *ChangeAddress4* and which is a copy of *ChangeAddress1*. In addition to *list.xhtml* (which is unchanged) there are three new JSF pages, and the entry of an an address occurs as a wizard with these three JSF pages:



index.xhtml is now reduced to the following very simple page:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   xmlns:f="http://xmlns.jcp.org/jsf/core">
 <h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <title>Change address</title>
  <h:outputStylesheet library="css" name="styles.css"/>
 </h:head>
 <h:body>
  <h1>Change address</h1>
  <h:form>
   <p><h:commandLink value="Create new address"
    action="#{indexController.page1()}" /></p>
   <p><h:commandLink value="Show addresses" action="list" /></p>
  </h:form>
 </h:body>
</html>
```

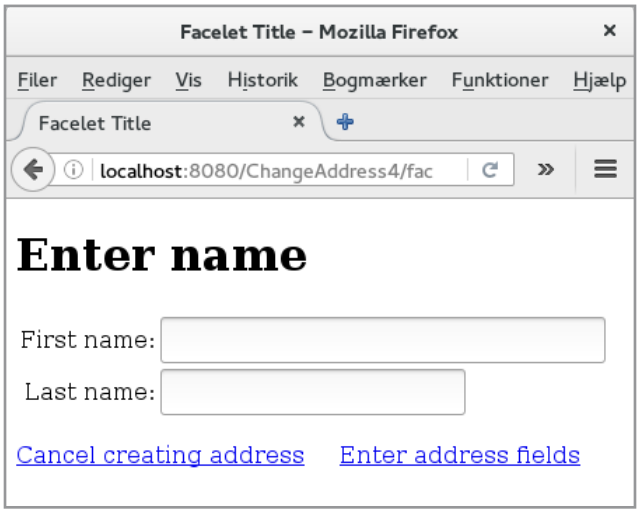
that opens the following window:

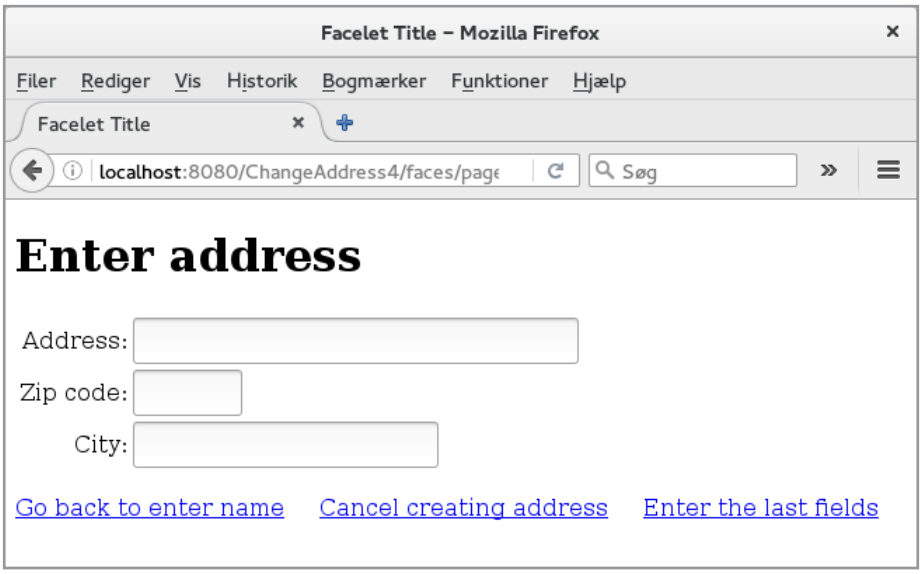


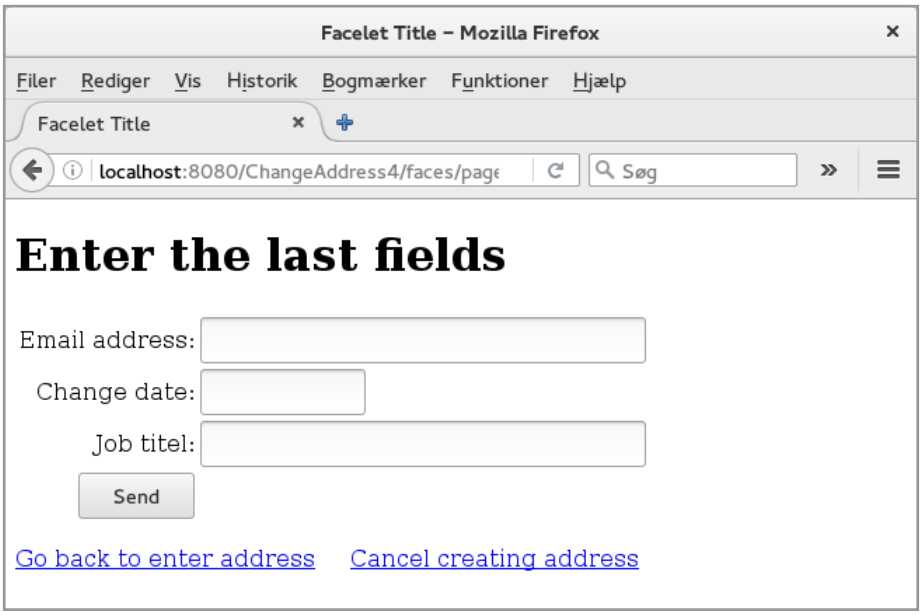
Clicking on the top link will give you a page for entering the name, and the form of the *ChangeAddress1* project is generally divided into three pages, as shown below. The three pages are called

- *page1.xhtml*
- *page2.xhtml*
- *page3.xhtml*

The entity class *Address* is unchanged from previously, but a simple wrapper class *Person* (a managed bean) is defined and all you need to note is that it is a named bean defined as *ConversationScoped*:




Facelet Title – Mozilla Firefox
Filer Rediger Vis Historik Bogmærker Funktioner Hjælp
Facelet Title
localhost:8080/ChangeAddress4/faces/page
Søg
Enter address
Address:
Zip code:
City:
Go back to enter name
Cancel creating address
Enter the last fields




```
package changeaddress.beans;

import javax.inject.Named;
import javax.enterprise.context.ConversationScoped;
import java.io.Serializable;

import changeaddress.models.Address;

@Named(value = "person")
@ConversationScoped
public class Person implements Serializable
{
 private Address address;

 public Person()
 {
  address = new Address();
 }

 public Address getAddress()
 {
  return address;
 }

 public void setAddress(Address address)
 {
  this.address = address;
 }
}
```

The code for the three new JSF pages is simple and I do not want to show it here, but the new is in *IndexController*, which has been changed. The code is as shown below, but I have not shown *get* and *set* methods since they are as before:

```
package changeaddress.beans;

import java.util.*;
import javax.inject.*;
import javax.enterprise.context.*;
import java.io.Serializable;
import javax.persistence.*;
import javax.transaction.*;
import javax.annotation.*;

import changeaddress.models.*;
import javax.persistence.criteria.CriteriaQuery;

@Named(value = "indexController")
@RequestScoped
public class IndexController implements Serializable
{
 @Inject
 private Conversation conversation;
 @Inject
 private Person person;

 @PersistenceUnit
 EntityManagerFactory emf;
 @PersistenceContext
 EntityManager em;
 @Resource
 UserTransaction utx;

 private List<Address> persons = new ArrayList();

 public IndexController()
 {
 }

 …

 public String cancel()
 {
  conversation.end();
  return "index";
 }
```

```
public String page1()
{
 conversation.begin();
 return "page1";
}

public List<Address> getPersons()
{
 try
 {
  CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
  cq.select(cq.from(Address.class));
  Query q = em.createQuery(cq);
  return persons = q.getResultList();
 }
 catch (Exception ex)
 {
 }
 return persons;
}

public String add()
{
```

```
  try
  {
   utx.begin();
   em.persist(person.getAddress());
   utx.commit();
   conversation.end();
   return "index";
  }
  catch (Exception ex)
  {
   try
   {
    utx.rollback();
   }
   catch (Exception e)
   {
   }
  }
  return "";
 }
}
```

Here you should note that the *context* is now *RequestScoped*. In addition, note the two methods *page1()* and *cancel()*. The first is called from *index.xhtml*, if you chooses to create a new address. It calls the method *conversation.begin()*, which starts a new conversation. The method *cancel()* is called from one of the three editing pages if you click *Cancel*, and it is important that it performs the method *conversation.end()* that completes the conversation that is in progress. This means that the container remove the bean. The method *getPersons()* is unchanged from previously, while the method *add()* has changed a bit. Most importantly, after an address is stored in the database, a *conversation.end()* is executed.

In order for it all to work, the class must instantiate a *Conversation* object, which occurs when using injection at the start of the class. A *Person* object must also be created, which also occurs by injection.

Also note how the class instantiate objects to JPA in the same way as in the previous example. In fact, it also happens by injection, only other annotations are used.

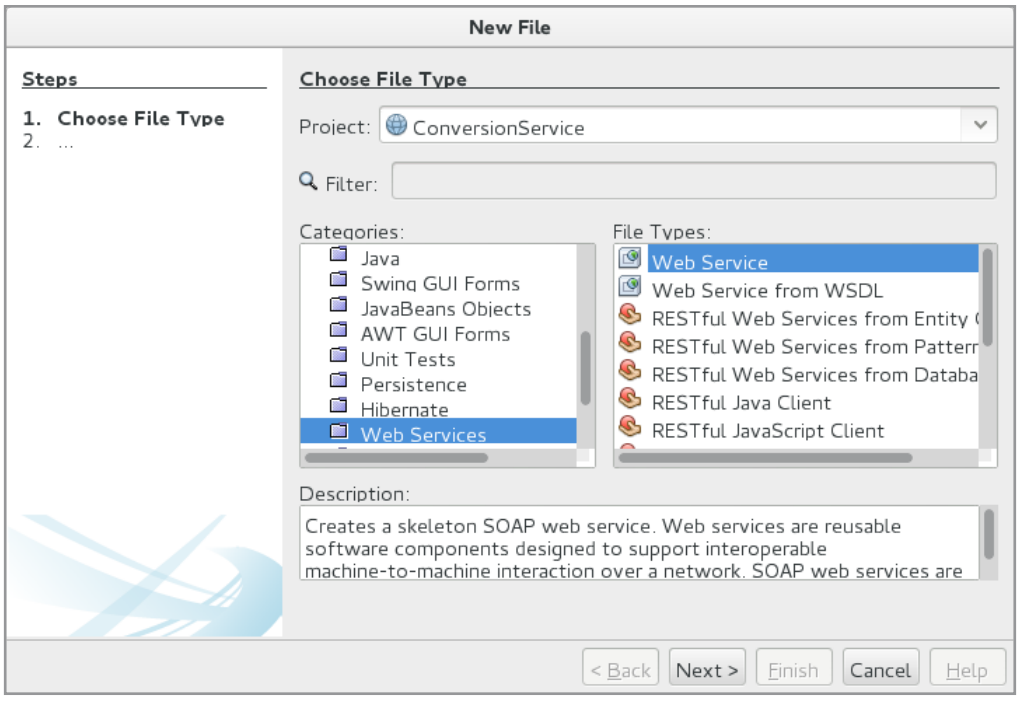
5 WEB SERVICES

A web service is a component that can be hosted on a web server and provides services to clients, and web services can to some extent be perceived as an alternative to enterprise java beans. Web services are characterized by being easy to develop and are platform independent. The latter means that communication between a web service and clients occurs as XML and after a protocol called SOAP, and thus exclusively as text. Therefore, the use of web services is perceived as a secure technology, and the price is that encoding of messages such as XML may cost a little on performance. A web service and its services must be defined to clients with regard to parameters and return values, and it is done using an XML document called a *Web Service Definition Language* (WSDL) document. The biggest challenge in writing web services is to write this document, which is quite complex (difficult and comprehensive syntax), but here NetBeans helps as it generates it all, and you can actually write web services without knowing anything about WSDL.

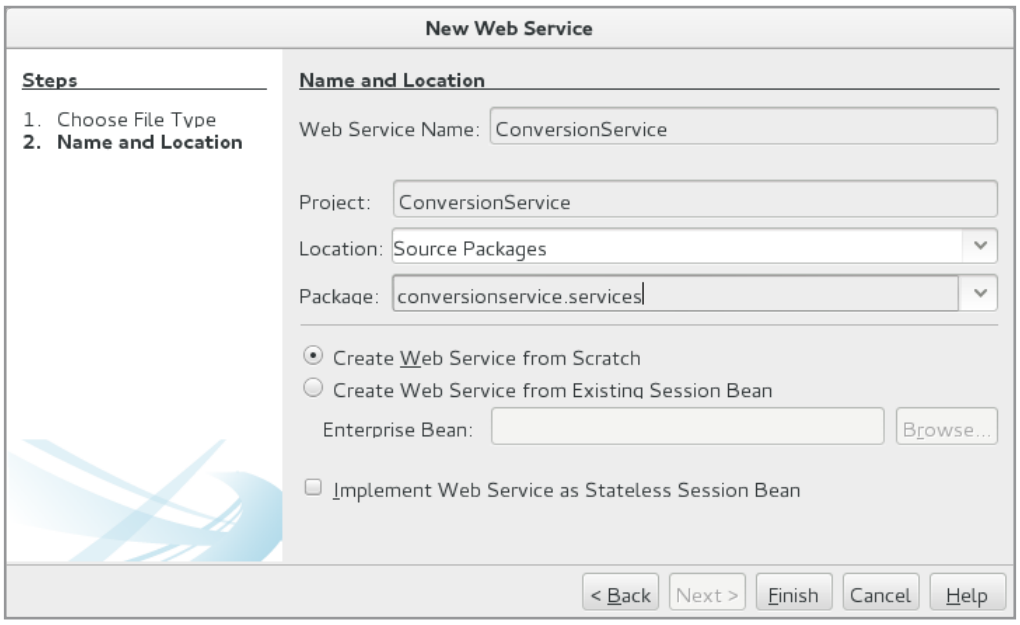
I want to start with a simple web service that has two services:

- *double celsiusToFahrenheit(double value)* which converts from fahrenheit to celsius
- *double fahrenheitToCelsius(double value)* which converts from celsius to fahrenheit

I start with a web application, which I have called *ConversionService*, and it just has to be an empty web application. Then I add a *Web Service*



and in the following window I have to enter a name (as here is *ConversionService*) and choose a package:



When I click *Next*, NetBeans creates the following class:

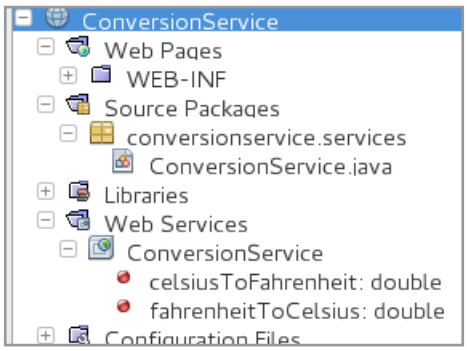
```
package conversionservice.services;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService(serviceName = "ConversionService")
public class ConversionService
{
 @WebMethod(operationName = "hello")
 public String hello(@WebParam(name = "name") String txt)
 {
  return "Hello " + txt + " !";
 }
}
```

Note which annotations are used, and that NetBeans creates a single service – which you should of course delete. This can be done directly, but clicking the *Design* button in the toolbar will give you a wizard that can be used to generate the finally code:

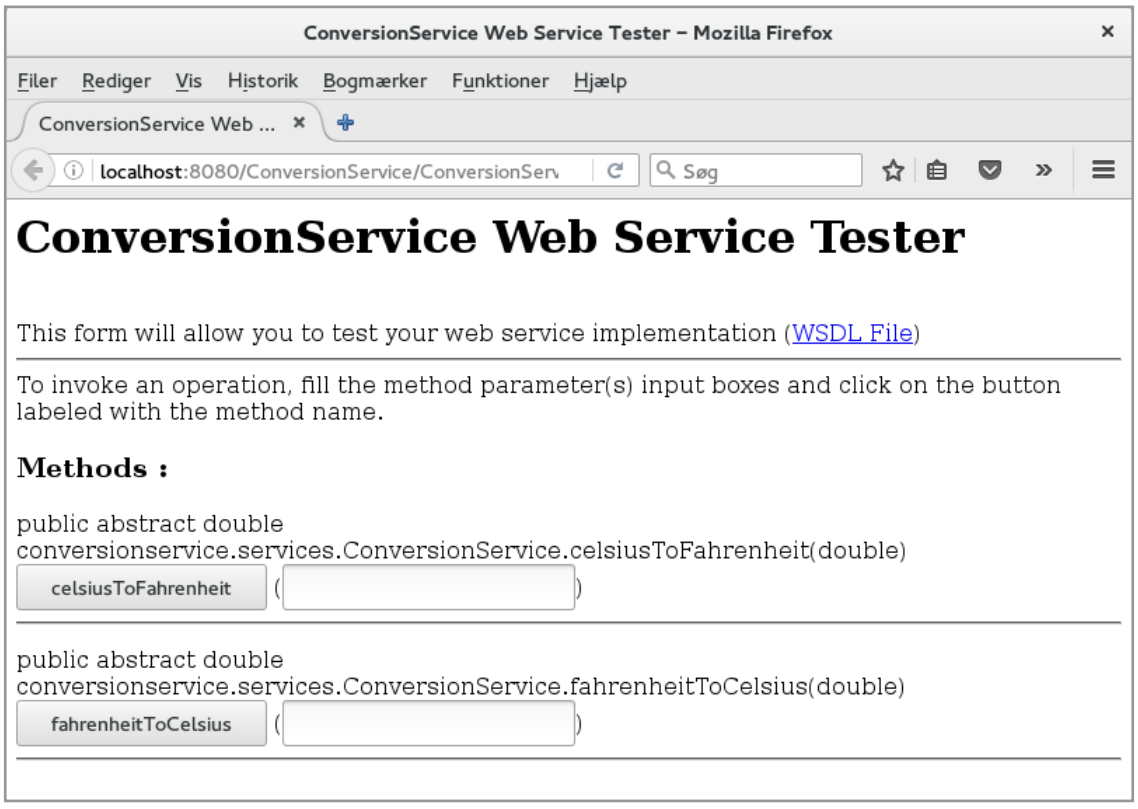
```
package conversionservice.services;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService(serviceName = "ConversionService")
public class ConversionService
{
 @WebMethod(operationName = "celsiusToFahrenheit")
 public double celsiusToFahrenheit(@WebParam(name = "value") double value)
 {
  return 1.8 * value + 32;
 }
 @WebMethod(operationName = "fahrenheitToCelsius")
 public double fahrenheitToCelsius(@WebParam(name = "value") double value)
 {
  return (value - 32) / 1.8;
 }
}
```

The biggest and only advantage of using the wizard is that NetBeans inserts the correct annotations.

Then you can build the project and choose deploy. In the NetBeans project tab, a *Web Servers* element is created:

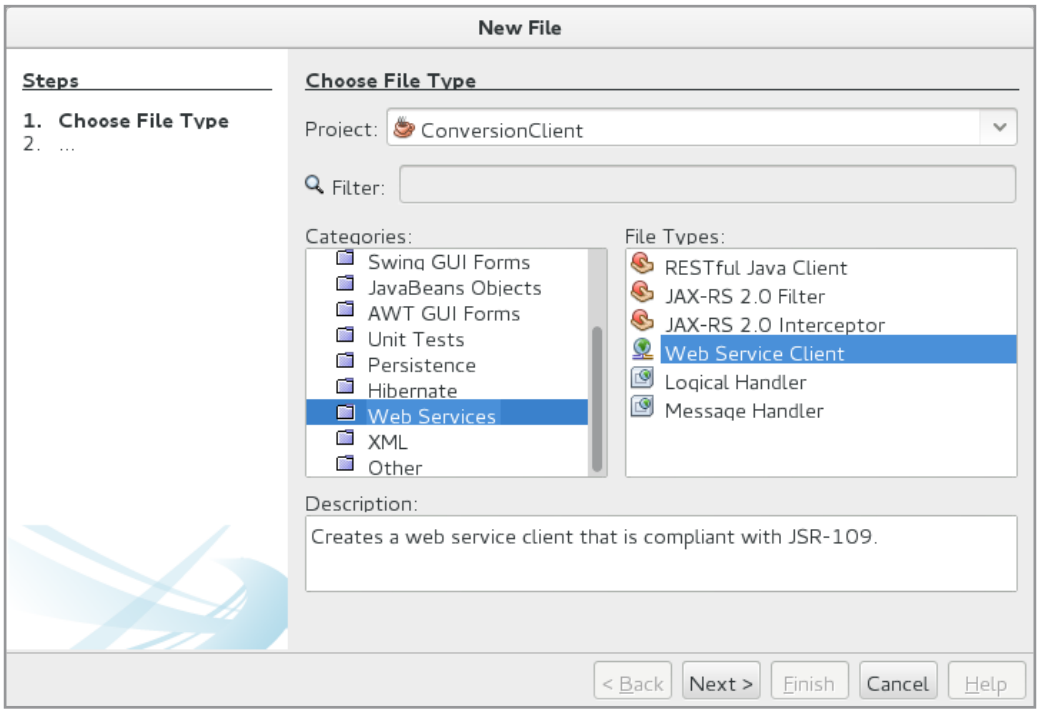


If you right-click *ConversationService* and choose *Test Web Service*, the browser opens a window to test the service:

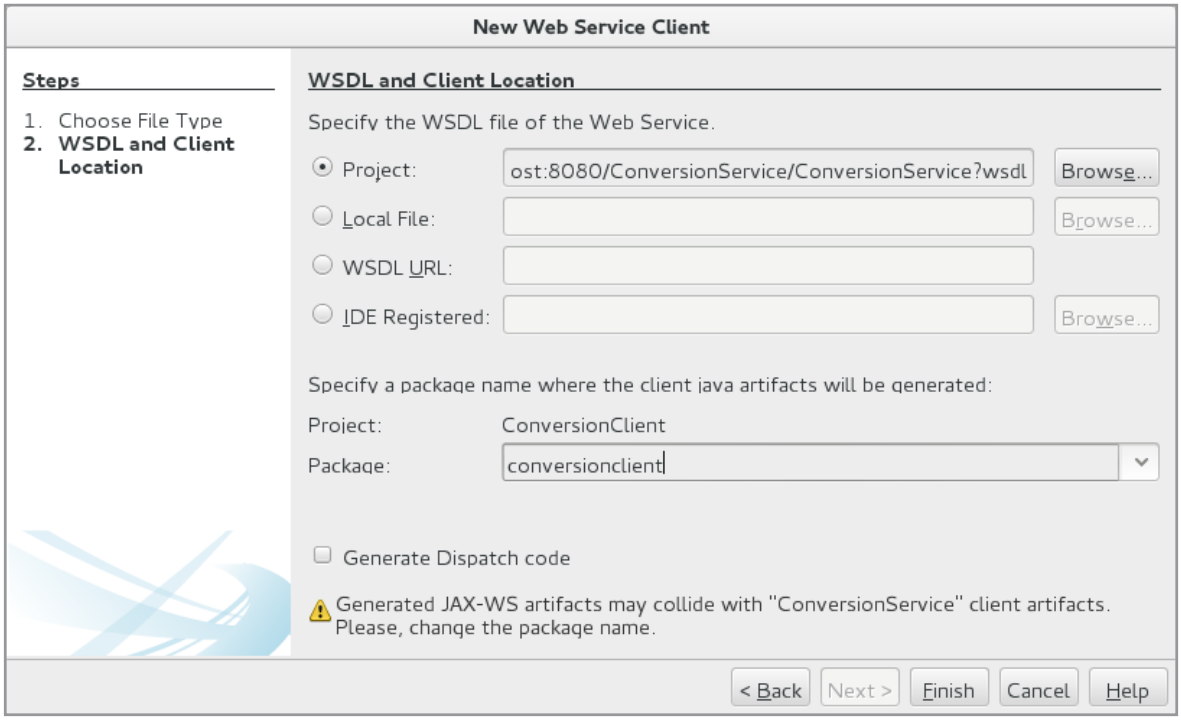


Here you should especially note the link *WSDL File*. Clicking on it will display the WSDL code that defines the service in question.

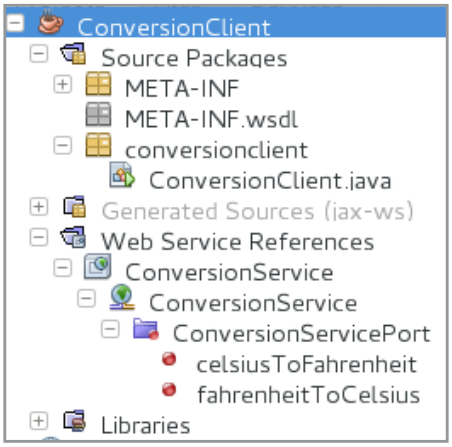
After the service is tested, a client must be written, and this time it should be a simple console application called *ConversionClient*. After I've created the application, I've added a *Web Service Client*:


New File
Steps
1. Choose File Type
2. ...
Choose File Type
Project:
ConversionClient
Filter:
Categories:
Swing GUI Forms
JavaBeans Objects
AWT GUI Forms
Unit Tests
Persistence
Hibernate
Web Services
XML
Other
File Types:
RESTful Java Client
JAX-RS 2.0 Filter
JAX-RS 2.0 Interceptor
Web Service Client
Logical Handler
Message Handler
Description:
Creates a web service client that is compliant with JSR-109.
< Back
Next >
Finish
Cancel
Help

In the next screen, under *Project*, select the current web service, and NetBeans will simply insert its URL. Finally, enter a package:



NetBeans now creates a reference to the web service and you can use the services it provides:



To use a service, you has write a complex statement, but you can simply drag a service into the client's code, and NetBeans creates the required one. Below is the completed code for the client:

```
package conversionclient;

public class ConversionClient
{
 public static void main(String[] args)
 {
  System.out.println("30 Celsius = " + celsiusToFahrenheit(30) + " Fahrenheit");
  System.out.println(
   "100 Fahrenheit = " + fahrenheitToCelsius(100) + " Celsius");
 }

 private static double celsiusToFahrenheit(double value)
 {
  conversionclient.ConversionService_Service service =
   new conversionclient.ConversionService_Service();
  conversionclient.ConversionService port = service.getConversionServicePort();
  return port.celsiusToFahrenheit(value);
 }

 private static double fahrenheitToCelsius(double value)
 {
  conversionclient.ConversionService_Service service =
   new conversionclient.ConversionService_Service();
  conversionclient.ConversionService port = service.getConversionServicePort();
  return port.fahrenheitToCelsius(value);
 }
}
```

You should especially note the two methods that NetBeans has created, but otherwise the program is ready for use:

```
30 Celsius = 86.0 Fahrenheit
100 Fahrenheit = 37.77777777777778 Celsius
```

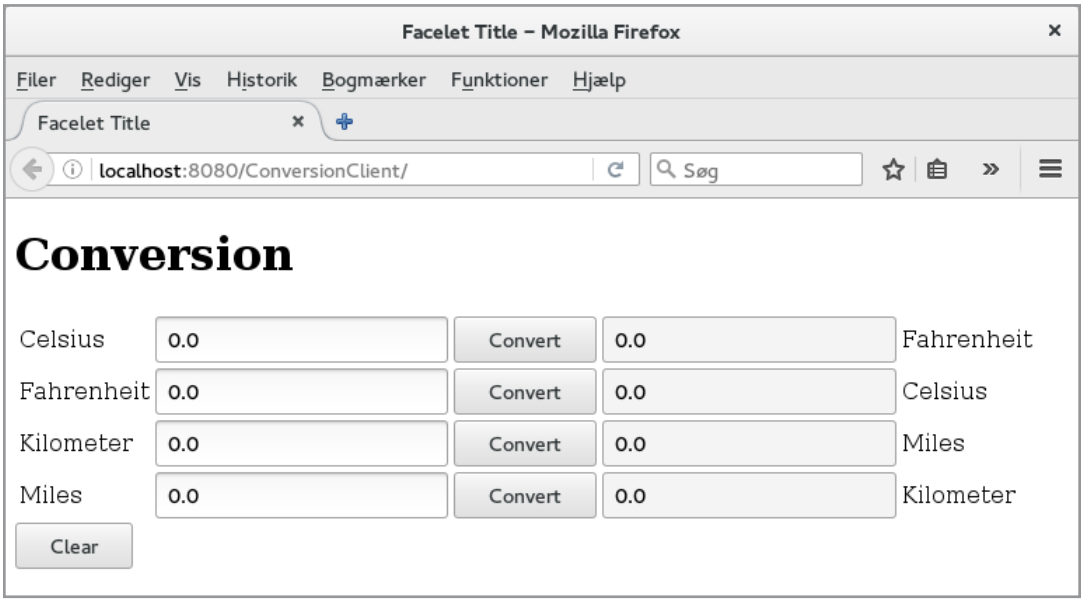
EXERCISE 4

Start by creating a copy of the *ConversionService* project. You must then expand the corresponding web service with two new services:

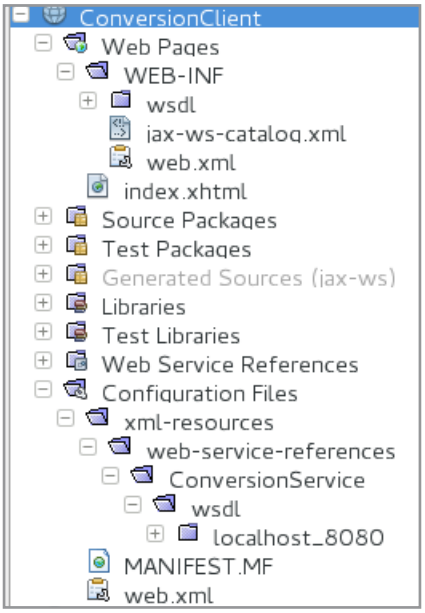
- *double kilometerToMiles(double value)*
- *double milesToKilometer(double value)*

which converts kilometers to and from miles.

When you have written the service and deployed it, write a client program that you can call *ConversionClient*, but this time it should be a web application. If the application is opened in the browser, it must open the following window:

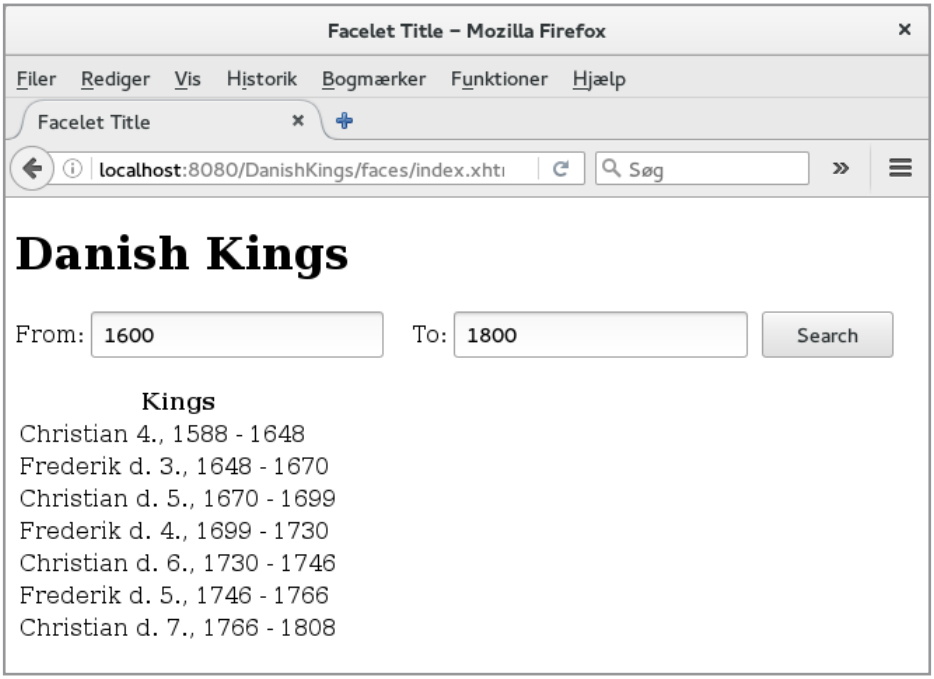


The above form should have a backing bean that uses your web service. In principle, it works exactly the same as in the introductory example, but with the difference that you should copy the folder *localhost_8080* to the *wsdl* folder under WEB-INF:



EXERCISE 5

In this exercise, you must write a web application where you can search for Danish kings:



You can call the project *DanishKings*, and the difference compared to before should be that the search this time should be done by a web service.

Start by creating a web application project that you can call *KingsServices* and add a web service called *KingService*. Add a simple model class

```
package kings.services;

public class King
{
 private String name;
 private int from;
 private int to;
```

The web service in question must have two services:

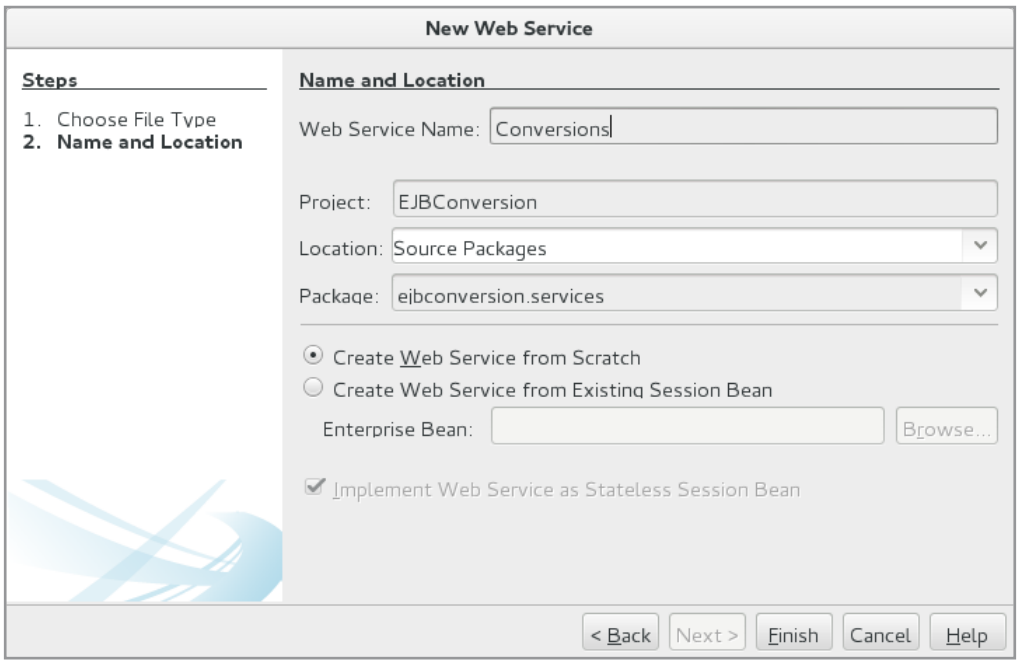
```
@WebMethod(operationName = "getKing")
public List<King> getKing(@WebParam(name = "year") int year)
{
}

@WebMethod(operationName = "getKings")
public List<King> getKings(@WebParam(name = "year1") int year1,
       @WebParam(name = "year2") int year2)
{
}
```

where the first will return the kings who ruled a certain year, while the other will return the kings whose reign overlaps a certain interval. Names of the Danish kings and reigns can be stored as constants in the program in the same way as in previous examples.

5.1 AN EJB AS A WEB SERVICE

As shown above, a web service is not much more than a usual Java class decorated with a few annotations and it all wrapped up in a web application. It is also possible to use a stateless session bean as a web service. It allows for using a stateless EJB from languages other than Java, as well as using other properties of enterprise java beans such as transactions. In fact, you can do it in two ways by either creating a web service in an EJB or by embedding an EJB as a web service.

As an example, I have set up an *EJB Module* project, which I have called *EJBConversion*. For this project I have added a web service called *Conversion*:



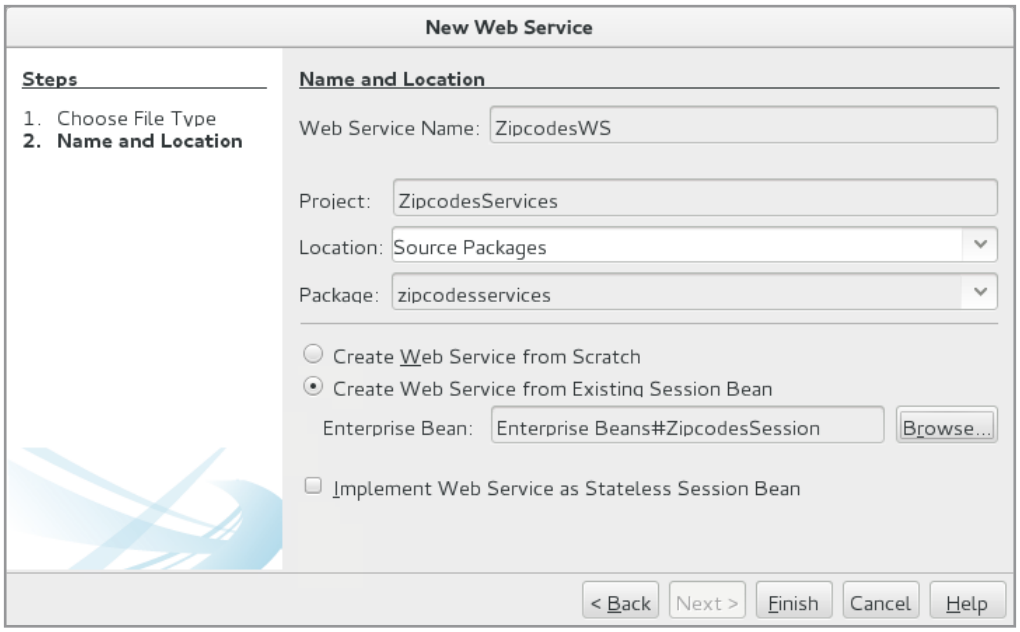
Note that *Create Web Service from Scratch* has been selected, and the bottom check box has been ticked. After clicking *Finish*, a web service has been created, where there is primarily one difference, namely, that it is a stateless session bean. The completed service is shown below:

```
package ejbconversion.services;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.ejb.Stateless;

@WebService(serviceName = "Conversions")
@Stateless()
public class Conversions
{
 @WebMethod(operationName = "celsiusToFahrenheit")
 public double celsiusToFahrenheit(@WebParam(name = "value") double value)
 {
  return 1.8 * value + 32;
 }

 @WebMethod(operationName = "fahrenheitToCelsius")
 public double fahrenheitToCelsius(@WebParam(name = "value") double value)
 {
  return (value - 32) / 1.8;
 }
}
```

Then the project can be built and deployed in the usual way. The *ConversionProgram* project is identical to *ConversionClient*, only with the difference that another web service is used, as above, which is a web service hosted as an enterprise java bean.

As the last example, I will write a web service that uses an existing EJB, and I will use *ZipcodesEJB*, which performs CRUD operations on the table *zipcode* in the database *padata*. The EJB concerned offers 4 services, and the goal is to provide these services with a web service.

The starting point is an empty web application called *ZipcodesServices*. As a first step, I have put a reference to the jar file *ZipcodesRemote*, which is the remote interface for *ZipcodesEJB*. Next, I have added a web service called *ZipcodesWS* to the project:



Here you should especially note that *Create Web Service from Existing Session Bean* has been selected, and I have then browsed me to that particular bean. When you click *Finish*, NetBeans creates the following web service:

```
package zipcodesservices;

import java.util.List;
import javax.ejb.EJB;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import zipcodes.beans.ZipcodeRemote;
import zipcodes.beans.ZipcodesSessionRemote;

@WebService(serviceName = "ZipcodesWS")
public class ZipcodesWS
{
 @EJB
 private ZipcodesSessionRemote ejbRef;

 @WebMethod(operationName = "create")
 public boolean create(@WebParam(name = "obj") ZipcodeRemote obj)
 {
  return ejbRef.create(obj);
 }
```

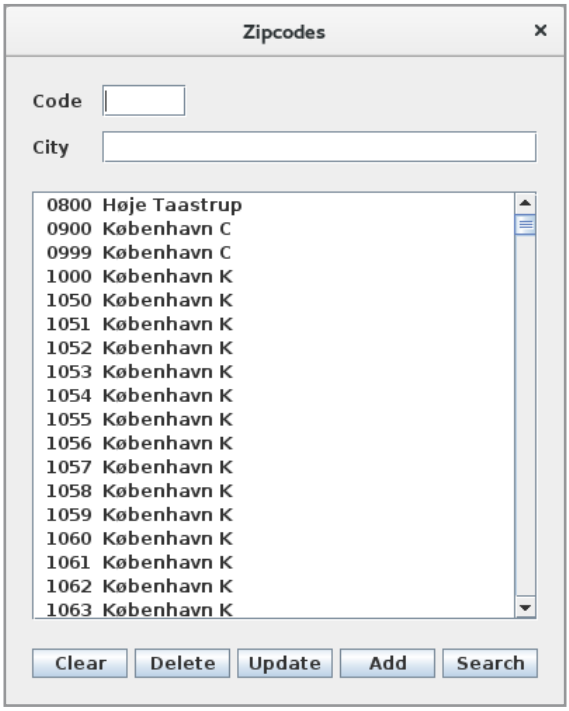
```
 @WebMethod(operationName = "update")
 public boolean update(@WebParam(name = "obj") ZipcodeRemote obj)
 {
  return ejbRef.update(obj);
 }

 @WebMethod(operationName = "remove")
 public boolean remove(@WebParam(name = "code") String code)
 {
  return ejbRef.remove(code);
 }

 @WebMethod(operationName = "search")
 public List<ZipcodeRemote> search(@WebParam(name = "code") String code,
  @WebParam(name = "City") String City)
 {
  return ejbRef.search(code, City);
 }
}
```

and as you can see, it's not much more than a wrapper for *ZipcodesEJB*. The service is then completed, can be translated and hosted on the server.

I have then written a client, which is a web application called *ZipcodesProgram*. It is basically the same as *ZipcodesClient*, and the difference is that a *Web Service Client* has been added and the program's named bean is changed a bit. I do not want to show the code here.

EXERCISE 6

You must write a program that corresponds to the above *ZipcodesProgram* which uses the web service *ZipcodesWS* to perform CRUD operations on the table *zipcode*, but instead of a web application, it must be a GUI application, which you can call *Zipcodes*. The user interface should be as shown below:

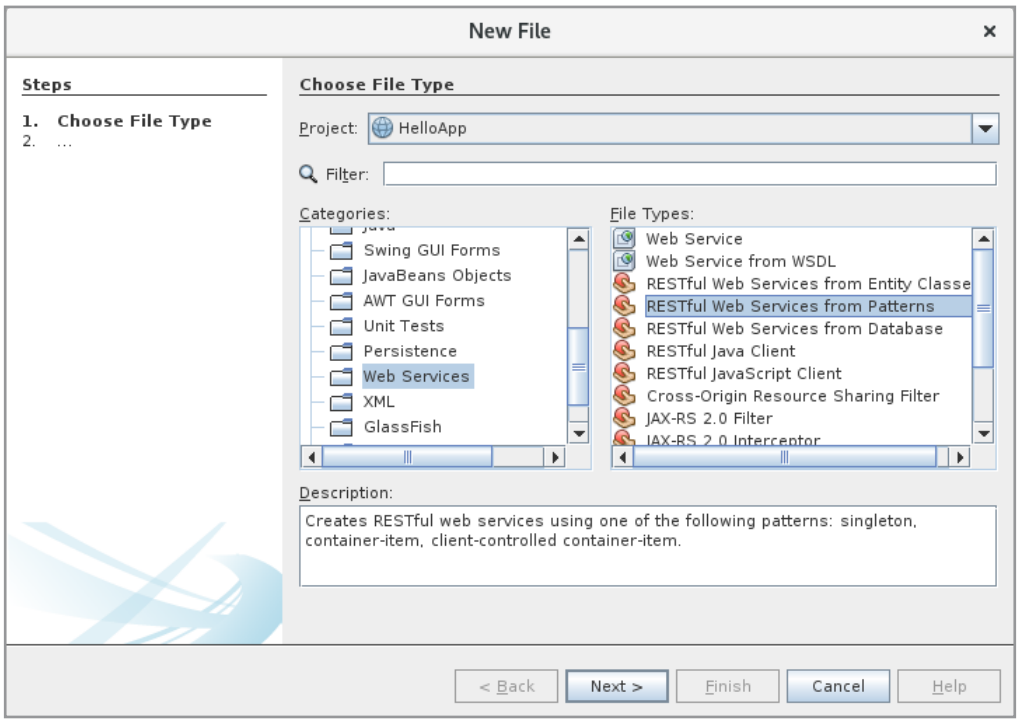


6 REST WEB SERVICES

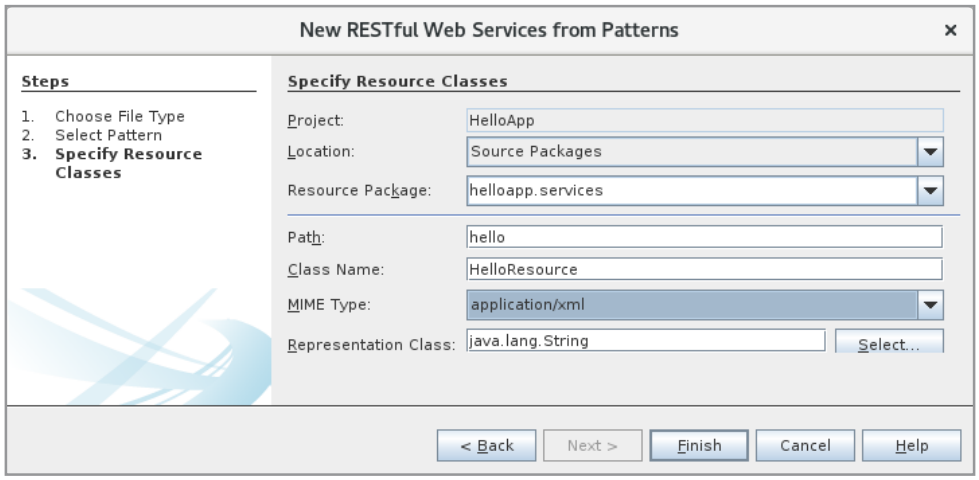
REST stands for *Representational State Transfer* and is also called for a *RESTful* web services. Like other web services, they can be perceived as resources made available and identified by an URI, and they are characterized by being easy to develop and are specifically intended as a frontend for a database table. RESTful web services are generally very flexible, and the communication between client and service can be done in several ways, but generally, data is sent either as XML or JSON. A web service must support at least one of four HTTP methods:

1. GET, which is used to retrieve an existing resource
2. POST, which is used to update an existing resource
3. PUT, which is used to create a new resource
4. DELETE, which is used to delete an existing resource

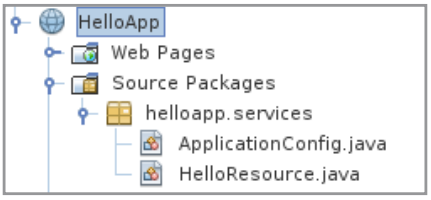
I want to start with a simple "hello" service and I have set up a usual (blank) web application project, which I have called *HelloApp*. After I've created the project, I've added a *RESTful Web Services from Patterns*:



and after I click *Next* twice, I come to the following window:



where I have entered *Resource Package*, *Path* and *Class Name*. You should note that the MIME type is XML. When I then click *Finsh*, NetBeans creates two files:



Here's the last one is my RESTful web service. The code is the following where I deleted comments, as well as entered code in the method *getXml()*:

```
package helloapp.services;

import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PUT;
import javax.ws.rs.core.MediaType;

@Path("hello")
public class HelloResource
{
 @Context
 private UriInfo context;

 public HelloResource()
 {
 }

 @GET
 @Produces(MediaType.APPLICATION_XML)
 public String getXml()
 {
  return
    "<message>\n"
     + "<hello>Hello from</hello>\n"
     + "<firstName>Ragnar</firstName>\n"
     + "<lastName>Lodbrog</lastName>\n"
     + "<job>Viking</job>\n"
    + "</message>\n";
 }

 @PUT
 @Consumes(MediaType.APPLICATION_XML)
 public void putXml(String content)
 {
 }
}
```

You must note the different annotations. Here, *@Path* specifies the name for which the current service (resource) will be known. *@GET* defines a method that is performed if an HTTP GET occurs from a browser, and similarly, *@PUT* defines a method for an HTTP

PUT. Note that the first method is decorated with *@Produces*, which indicates that the method returns XML while the other is decorated with *@Consumes*, which indicates that the method interprets the content of the parameter as XML. In this case, the last method is empty, while the first returns some XML.

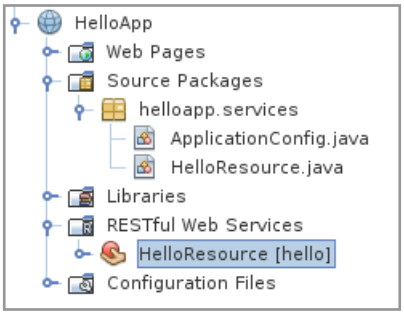
The container must know what the resource is called and that is the purpose of the second class:

```
package helloapp.services;

import java.util.Set;
import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application
{
 @Override
 public Set<Class<?>> getClasses() {
  Set<Class<?>> resources = new java.util.HashSet<>();
  addRestResourceClasses(resources);
  return resources;
 }

 private void addRestResourceClasses(Set<Class<?>> resources)
 {
  resources.add(helloapp.services.HelloResource.class);
 }
}
```

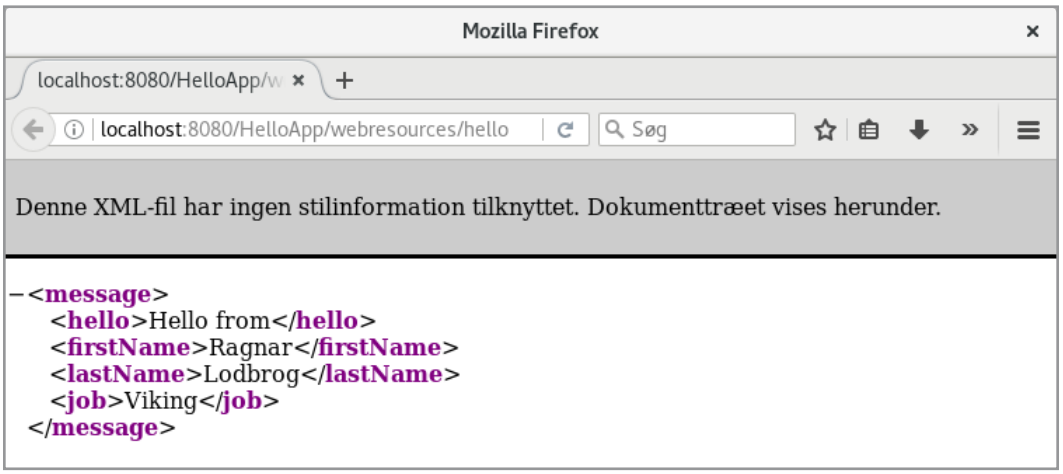
It is a class inheriting *Application*, and the class is decorated with *@ApplicationPath*, which specifies a name (here it is webresources) that is included in the resource's URI. The last method adds the actual resource object to a collection (here it is a *HashSet*). The result of all that is, the resource's URI is

```
http://localhost:8080/helloapp/webresources/hello
```

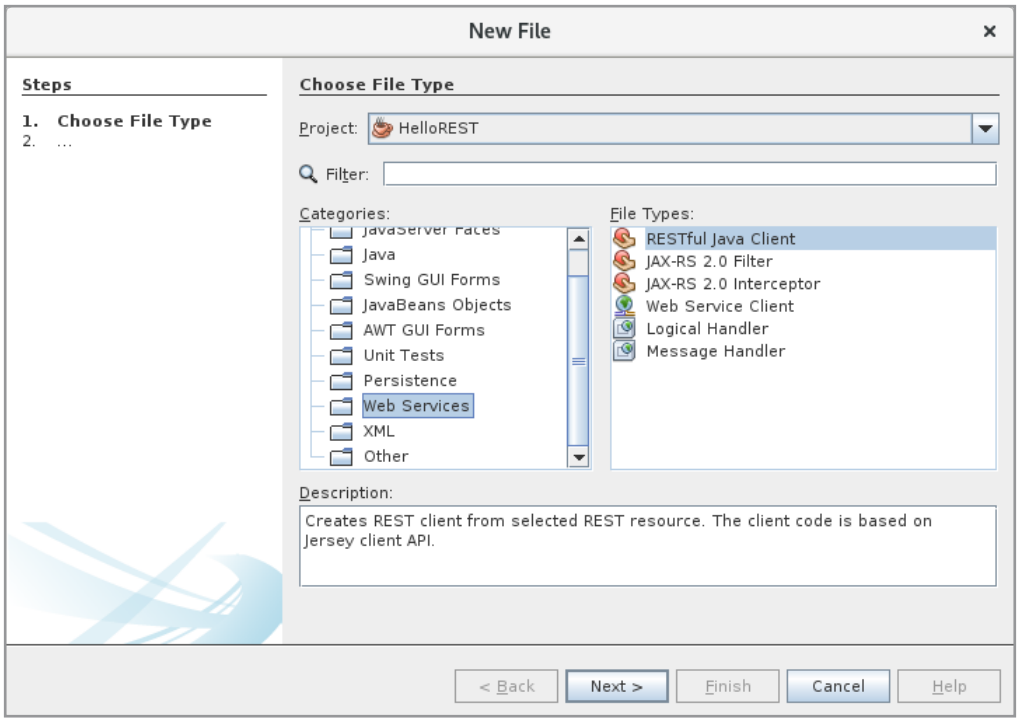
If you translate the project and perform a deploy, the resource is ready for use. An *RESTful Web Services* item is created and if you right click on *HelloResource [hello]*:



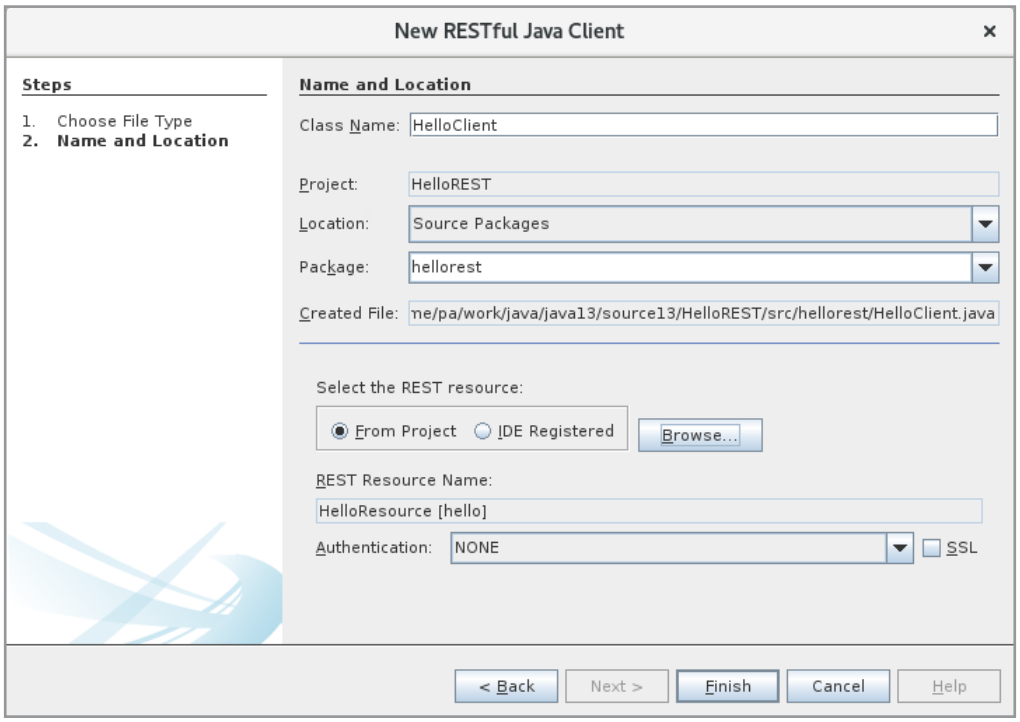
you can you choose *Test Resource URI*, after which the browser opens the following window:



The result is that the method *getXml()* is performed and the method has sent a response to the browser. As a next step, I will write a client that uses the resource and it should be a simple console application. I have therefore created a Java Application project called *HelloREST*. For this project I have added a *RESTful Java Client*:



When you click Next, you get the following window:



Here I have entered *Class Name*, selected a package, and used the *Browse* button set a reference to the resource. Note that *Authentication* is NONE (which is default). After clicking *Finish*, NetBeans creates the following class that represents the service to the client:

```
package hellorest;

import javax.ws.rs.ClientErrorException;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.WebTarget;

public class HelloClient
{
 private WebTarget webTarget;
 private Client client;
 private static final String BASE_URI =
  "http://localhost:8080/HelloApp/webresources";

 public HelloClient()
 {
  client = javax.ws.rs.client.ClientBuilder.newClient();
  webTarget = client.target(BASE_URI).path("hello");
 }

 public String getXml() throws ClientErrorException
 {
  WebTarget resource = webTarget;
  return resource.request(javax.ws.rs.core.MediaType.APPLICATION_XML).
   get(String.class);
 }

 public void putXml(Object requestEntity) throws ClientErrorException
 {
  webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_XML).
   put(javax.ws.rs.client.Entity.entity(requestEntity,
    javax.ws.rs.core.MediaType.APPLICATION_XML));
 }

 public void close()
 {
  client.close();
 }
}
```

I have then written the following *main()* metho

```
package hellorest;

public class HelloREST
{
 public static void main(String[] args)
 {
```

```
  HelloClient client = new HelloClient();
  System.out.println(client.getXml());
  client.close();
 }
}
```

and running the program is the result:

```
<message>
<hello>Hello from</hello>
<firstName>Ragnar</firstName>
<lastName>Lodbrog</lastName>
<job>Viking</job>
</message>
```

It may not be a lot, but you can see that the service in question is performed.

Looking at the above, it is quite simple to create and use a RESTful web service and write a client, but of course it is the problem that the result you receive is XML, and if you send something to the service, it should also be XML. However, converting XML to and from a custom type is easy, and it requires an annotation of the type, and that the type is a place

where it is known by both the service and the client program. A very simple solution to this problem is to define a class library containing the custom type, and in this case I have created a class library called *HelloRemote*, which contains a single class:

```
package helloremote.entities;

@javax.xml.bind.annotation.XmlRootElement
public class Message implements java.io.Serializable
{
 private String firstname;
 private String lastname;
 private String job;
 private String text;

 public Message()
 {
 }

 public Message(String firstname, String lastname, String job, String text)
 {
  this.firstname = firstname;
  this.lastname = lastname;
  this.job = job;
  this.text = text;
 }

 public String getFirstname()
 {
  return firstname;
 }

 public void setFirstname(String firstname)
 {
  this.firstname = firstname;
 }

 public String getLastname()
 {
  return lastname;
 }

 public void setLastname(String lastname)
 {
  this.lastname = lastname;
 }
```

```
 public String getJob()
 {
  return job;
 }

 public void setJob(String job)
 {
  this.job = job;
 }

 public String getText()
 {
  return text;
 }

 public void setText(String text)
 {
  this.text = text;
 }

 @Override
 public String toString()
 {
  return "firstname =\t" + firstname + "\nlastname =\t" + lastname +
    "\njob =\t\t" + job + "\ntext =\t\t" + text;
 }
}
```

It is nothing but a common bean and the only interesting thing is that the class is decorated with an annotation:

```
@javax.xml.bind.annotation.XmlRootElement
```

which ensures that an object can be automatically converted to and from XML. To use the type, the service needs to be changed:

```
package helloapp.services;

import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PUT;
import javax.ws.rs.core.MediaType;

import helloremote.entities.*;
```

```
@Path("hello")
public class HelloResource
{
 @Context
 private UriInfo context;

 private Message msg = new Message("Ragnar", "Lodbrog", "Viking", "Hello from");

 public HelloResource()
 {
 }

 @GET
 @Produces(MediaType.APPLICATION_XML)
 public Message getXml()
 {
  return msg;
 }

 @PUT
 @Consumes(MediaType.APPLICATION_XML)
 public void putXml(Message msg)
 {
```

```
  this.msg.setFirstname(msg.getFirstname());
  this.msg.setLastname(msg.getLastname());
  this.msg.setJob(msg.getJob());
  this.msg.setText(msg.getText());
 }
}
```

First, there should be a reference to the class library *HelloRemote*. Then an object of the type *Message* is created, and the return type for *getXml()* has been changed to *Message*. The method's *MediaType* is still XML. As a result, the object *msg* is encoded as XML before it is sent to a client. Similarly, *putXml()* is changed, so it has a *Message* object as parameter, and the method updates the *msg* object. This is of course pseudo, since the changes are not used for something – the message *msg* does not exist between two calls of the service.

After the service has been changed and after a deploy, the client program must also be updated. First, the class *HelloService* should be changed slightly. The resource is represented by an object of the type *WebTarget*, and if you on this object performs the method *request()* it calls the service's *get* method (here it is *getXml()* that is decorated with *@GET*). It tells to encode data as XML. Finally, with the *get()* method, you specify how the result should be converted and before it was to a *String*. It's changed to the type *Message*, and that's all that's necessary to change in the *HelloClient* class:

```
public String getXml() throws ClientErrorException
{
 WebTarget resource = webTarget;
 return resource.request(javax.ws.rs.core.MediaType.APPLICATION_XML).
  get(Message.class);
}
```

Finally, the *main()* method in *HelloREST* must be changed:

```
package hellorest;

import helloremote.entities.Message;

public class HelloREST
{
 public static void main(String[] args)
 {
  Message msg;
  HelloClient client = new HelloClient();
  System.out.println(msg = client.getXml());
  msg.setJob("Danish Viking");
  client.putXml(msg);
```

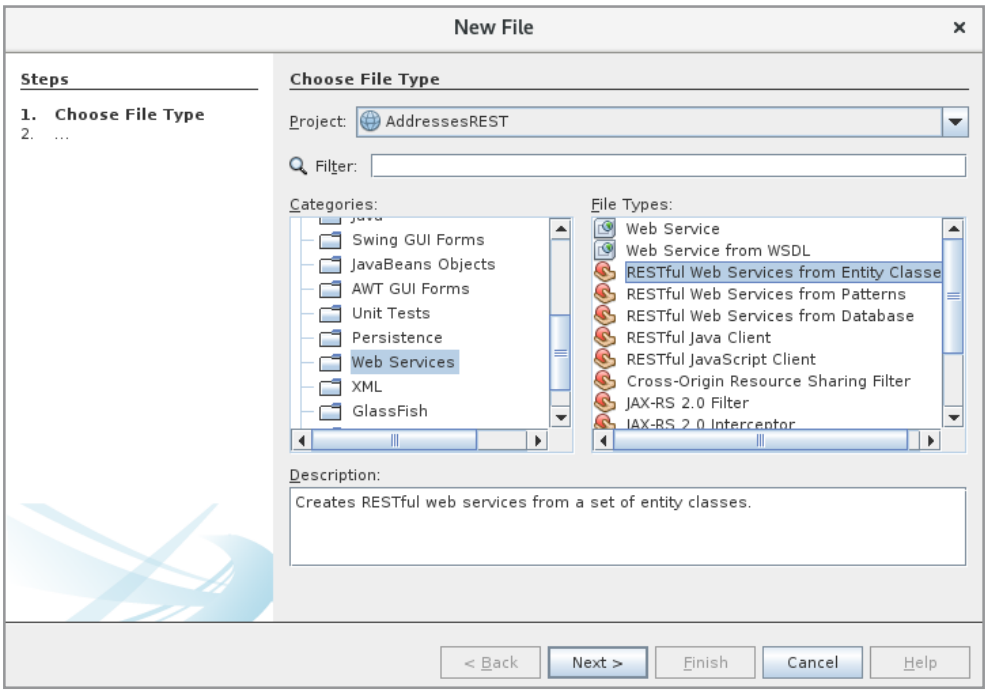
```
    client.close();
    System.out.println();
    System.out.println(msg);
  }
}
```

If the program should work as before, it's actually not necessary to change anything, and the changes concern only that I want to show how to use *putXml()* and that a *Message* object is automatically encoded as XML.

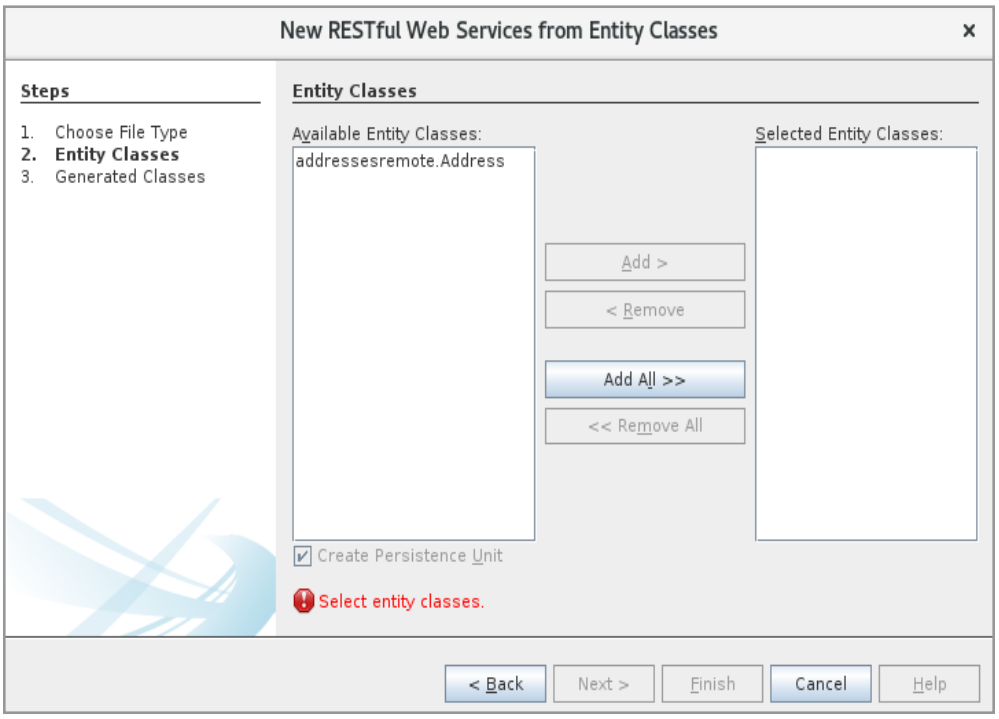
6.1 CHANGEADDRESS AGAIN

The above is a very simple RESTful web service and not a typical example. In this example, I want to show another version of the program *ChangeAddress*, where the difference should be that the program this time uses a RESTful web service to maintain the database.

I start by creating a *Class Library* project named *AddressesRemote*, and for this project I have added an *Entity Classes from Database*, which is an entity class for the table *address* in my MySQL database *addresses*. My library should not contain anything else and should be translated, so I it is ready for use. Then I created a new *Web Application* project named *AddressesREST*, and for this project I have added a web service as a *RESTful Web Service from Entity Classes*:



Then I have to choose an entity class (there is only one):



and in the following window I must select/enter a package where I have entered *addressrest.service*. The result is that NetBeans creates three files:

- *AbstractFacade.java*
- *AddressFacadeREST.java*
- *ApplicationConfig.java*

Here is the last identical to the config class from the first example, while the others define methods for maintaining the database using JPA. *AbstractFacade* is the base class for *AddressFacadeREST*, and I do not want to show the class here, but the code to the other is as follows:

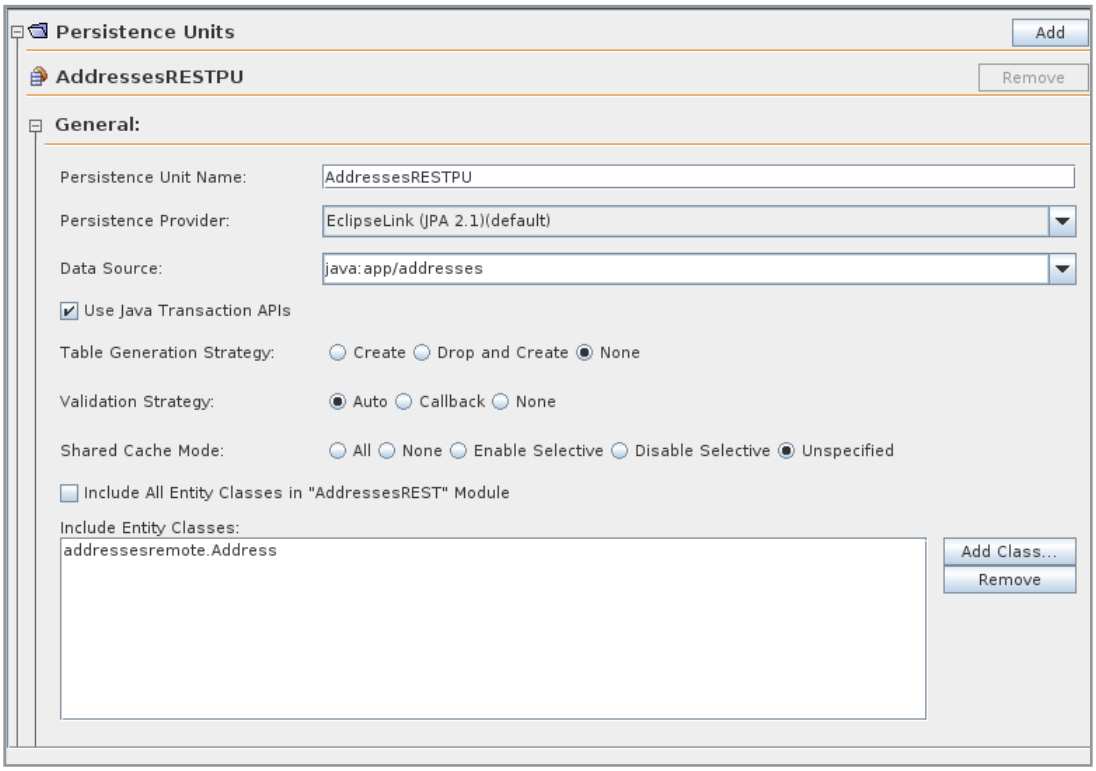
```
@Stateless
@Path("addressesremote.address")
public class AddressFacadeREST extends AbstractFacade<Address> {
 @PersistenceContext(unitName = "AddressesRESTPU")
 private EntityManager em;

 public AddressFacadeREST() {
  super(Address.class);
 }

 @POST
 @Override
 @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
 public void create(Address entity) {
  super.create(entity);
 }

 @PUT
 @Path("{id}")
 @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
 public void edit(@PathParam("id") Integer id, Address entity) {
  super.edit(entity);
 }

 @DELETE
 @Path("{id}")
 public void remove(@PathParam("id") Integer id) {
  super.remove(super.find(id));
 }

 @GET
 @Path("{id}")
 @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
```

```
 public Address find(@PathParam("id") Integer id) {
  return super.find(id);
 }

 @GET
 @Override
 @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
 public List<Address> findAll() {
  return super.findAll();
 }

 @GET
 @Path("{from}/{to}")
 @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
 public List<Address> findRange(
  @PathParam("from") Integer from, @PathParam("to") Integer to)
 {
  return super.findRange(new int[]{from, to});
 }

 @GET
 @Path("count")
 @Produces(MediaType.TEXT_PLAIN)
 public String countREST() {
  return String.valueOf(super.count());
 }

 @Override
 protected EntityManager getEntityManager() {
  return em;
 }
}
```

It is a RESTful web service and you can see that it is also a stateless session bean. The class creates an *EntityManager* using CDI, but the rest are methods that can be used by a client. You should note that there are methods decorated as:

- @POST to create an address
- @PUT to update an address
- @DELETE to delete an address

and that there are four methods decorated with *@GET*, which are used to request the database. The service is complete and can perform CRUD operations on the database. It does not look much, but the reason is that JPA takes care of everything (and of course because there is only one table). Note: If you have trouble deploying the service, check the configuration file:


Persistence Units
Add
AddressesRESTPU
Remove
General:
Persistence Unit Name:
AddressesRESTPU
Persistence Provider:
EclipseLink (JPA 2.1)(default)
Data Source:
java:app/addresses
Use Java Transaction APIs
Table Generation Strategy:
Create
Drop and Create
None
Validation Strategy:
Auto
Callback
None
Shared Cache Mode:
All
None
Enable Selective
Disable Selective
Unspecified
Include All Entity Classes in "AddressesREST" Module
Include Entity Classes:
addressesremote.Address
Add Class...
Remove

Here, you may need to add the entity class below *Include Entity Classes*. After that you should be able to deploy the project.

Next, a client must be written and I have started with a copy of the project *ChangeAddress1*, which I have called *ChangeAddress5*. Here I have deleted the model class *Address* and added a reference to the class library *AddressesRemote*. As the next step, I have added a *RESTful Java Client* named *AddressesClient*. It is similar to the corresponding class from the previous example, but you should note that methods have been created for both XML and JSON. Below I have shown some of the class, but only the methods my program needs:

```
package changeaddress.beans;

import javax.ws.rs.ClientErrorException;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.WebTarget;

public class AddressesClient
{
 private WebTarget webTarget;
 private Client client;
 private static final String BASE_URI =
  "http://localhost:8080/AddressesREST/webresources";

 public AddressesClient()
 {
  client = javax.ws.rs.client.ClientBuilder.newClient();
  webTarget = client.target(BASE_URI).path("addressesremote.address");
 }

 public void create_XML(Object requestEntity) throws ClientErrorException
 {
  webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_XML).
   post(javax.ws.rs.client.Entity.entity(requestEntity,
    javax.ws.rs.core.MediaType.APPLICATION_XML));
 }

 public void create_JSON(Object requestEntity) throws ClientErrorException
 {
  webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).
   post(javax.ws.rs.client.Entity.entity(requestEntity,
    javax.ws.rs.core.MediaType.APPLICATION_JSON));
 }

 public <T> T findAll_XML(Class<T> responseType) throws ClientErrorException
 {
```

```
   WebTarget resource = webTarget;
   return resource.request(javax.ws.rs.core.MediaType.APPLICATION_XML).
    get(responseType);
  }

  public <T> T findAll_JSON(Class<T> responseType)
 throws ClientErrorException
  {
   WebTarget resource = webTarget;
   return resource.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).
    get(responseType);
  }

  public void close() {
   client.close();
  }
 }
```

The class is similar to the corresponding method from the previous example. The constructor creates a reference to the resource that is represented by an object of the type *WebTarget*. This object can perform a request for the resource with one of the four HTTP methods. In the current example, it is GET to perform a request (return all addresses) and POST to create a new address.

Note the method *findAll_XML()*, which is generic. Here you must specify the type of the object that the method should return, and it is *List<Address>*. Since the result is XML, it must be parsed as a *List<Address>* object that occurs automatically, as shown in the previous example. However, it causes a problem as the *List<Address>* type is not decorated as

```
@XmlRootElement
```

It is therefore necessary to embed a *List<Address>* in a wrapper class:

```
package changeaddress.beans;

import java.util.List;
import javax.xml.bind.annotation.*;
import addressesremote.*;

@XmlRootElement(name="addresses")
@XmlAccessorType (XmlAccessType.FIELD)
public class Addresses
{
 @XmlElement(name = "address")
 private List<Address> addresses = null;
```

```
 public Addresses()
 {
 }

 public List<Address> getAddresses()
 {
  return addresses;
 }

 public void setAddresses(List<Address> addresses)
 {
  this.addresses = addresses;
 }
}
```

With this class in place, you can write the finished named bean, which is just a simple update of the existing class (I have not shown *get* and *set* methods):

```
package changeaddress.beans;

import addressesremote.Address;
import java.util.*;
```

```
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;

@Named(value = "indexController")
@SessionScoped
public class IndexController implements Serializable
{
 private Address person = new Address();
 private List<Address> persons = new ArrayList();

 public IndexController()
 {
 }

 public List getPersons()
 {
  try
  {
   AddressesClient client = new AddressesClient();
   Addresses wrapper = client.findAll_XML(Addresses.class);
   persons = wrapper.getAddresses();
   client.close();
  }
  catch (Exception ex)
  {
   persons = new ArrayList();
  }
  return persons;
 }

 public void add()
 {
  try
  {
   AddressesClient client = new AddressesClient();
   client.create_XML(person);
   client.close();
   person = new Address();
  }
  catch (Exception ex)
  {
  }
 }
}
```

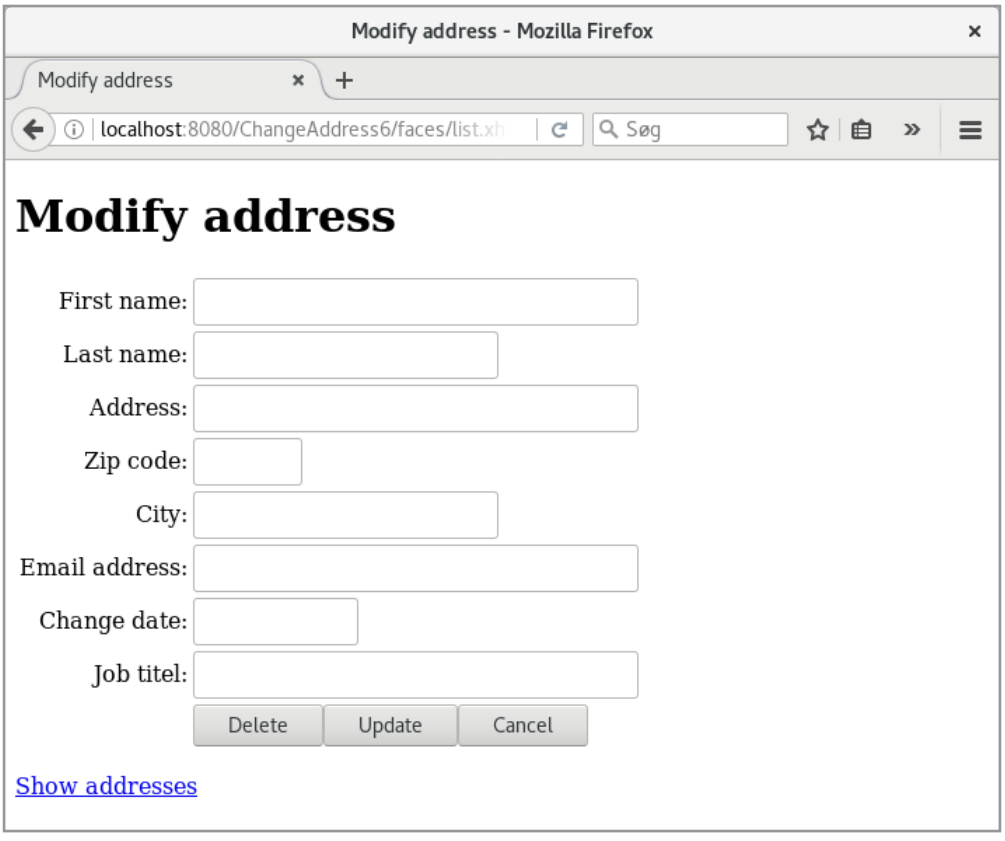
The two interesting methods are *getPersons()* and *add()* as the methods that uses the resource.

EXERCISE 7

You need to expand the above example so that you can also change and delete existing addresses, but so that you use the same web service as in the previous example. You can proceed as follows:

Start with a copy of the project *ChangeAddress5*, which you can call *ChangeAddress6*. It may require that you change the reference to *AddressesRemote*. After you have created the copy, you should test if the program still works properly.

1. Next, create a new page *edit.xhtml*, which is essentially the same as *index.xhtml* (see below).
2. You must modify page *list.xhtml*, so the name becomes a link that refers to the new page. When this page is opened, the form must be filled in with data for the current address.
3. You need to expand the controller *IndexController* so it has methods to click on a link in *list.xhtml* as well as to the three buttons in the new form.



Then everything should work.

7 SECURITY

When developing applications that are used over a network (web applications, enterprice applications, etc.), at the same time – or very often – there is a security issue that needs to be solved. It is a problem that I have overlooked in both this book and the two previous books. Basically it is about

1. that only allowed users may use a program or parts of an application and that all users should not have the same rights
2. that unauthorized users may not necessarily access the data sent over a network between users and servers

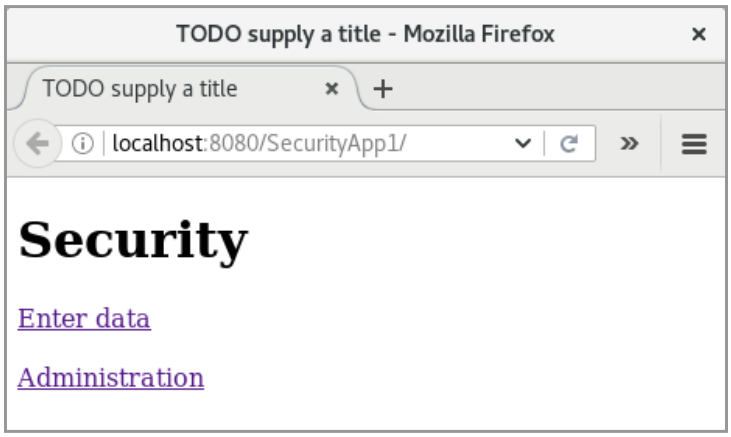
The first issue is solved by using user identification and passwords, while the other issue is solved by using encryption so nobody can read the data sent over the network. In principle, it sounds simple, but it's no matter what the many cyber attacks, as one hears in the press, also testifies. Perhaps it's actually impossible to completely secure an enterprise application, but you can do a lot, and more than often it is not done.

Many of the challenges associated with security are not so much because we do not know how, but more that they make everyday work difficult, and for that reason, many may be tempted to relax security policies. Studies actually shows that most security issues are due to how the programs are used, more than how the programs are made. In this chapter I will give an introduction to what safety matters there are as part of the development work, and mention that it does not help much unless users of IT solutions also take the security issue seriously.

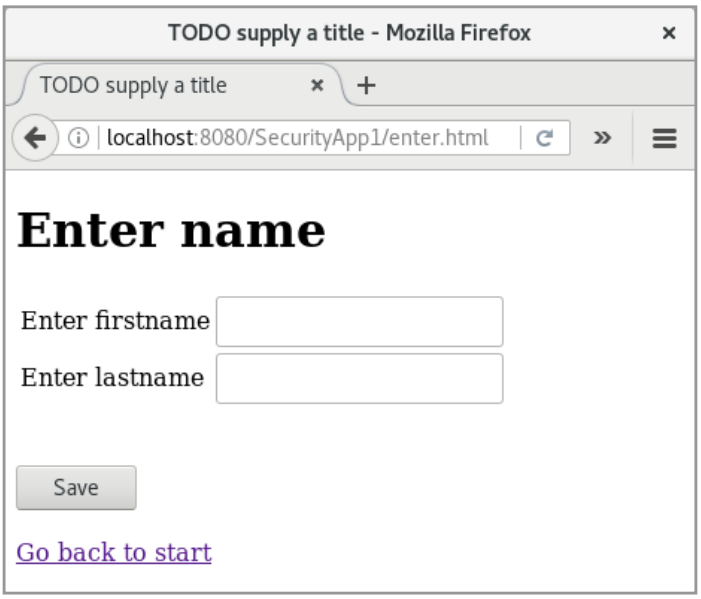
In turn, the challenges of encryption are the easiest to solve, as it is primarily a question of using a secure transmission line where data is sent encrypted, and the problem is solved by properly configuring the program. However, the authentication and user authorization issues can be solved by two sites, namely by the container (the application server), which is a configuration task, and by program itself, where it is the programmer who will solve the task.

7.1 THE DEMO APPLICATION

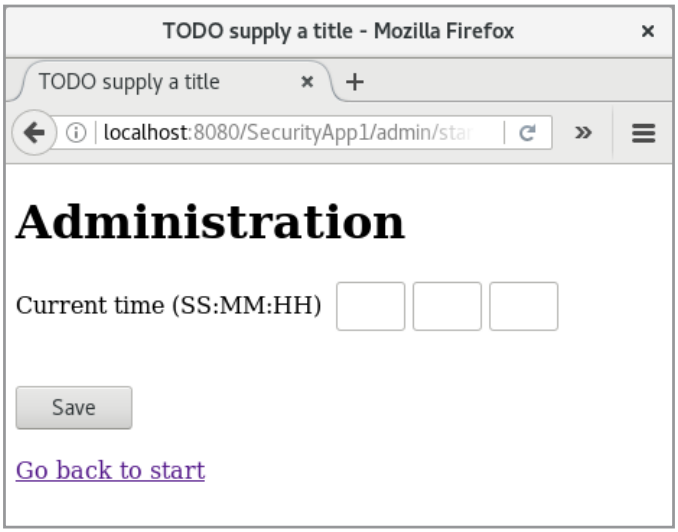
I use a simple web application called *SecurityApp1*. The application consists of only three simple HTML pages, where the *index.html* is the start page and opens the following window:



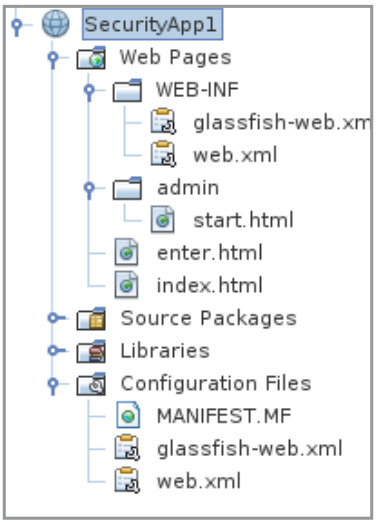
If you click on the top link, you get a window for entering a name (which is not saved):



If you go back to the start page and click the *Administration* link, you will get the following page where you can edit a time (which will not be saved):



The code is trivial and will not be shown here. Below are the project files, and here you should note that the page *start.html* is located in a folder *admin*, and that in WEB-INF is added a configuration file *glassfish-web.xml* (a Glassfish Descriptor file):


SecurityApp1
Web Pages
WEB-INF
glassfish-web.xm
web.xml
admin
start.html
enter.html
index.html
Source Packages
Libraries
Configuration Files
MANIFEST.MF
glassfish-web.xml
web.xml

7.2 CONTAINER MANAGED AUTHENTICATION AND AUTHORIZATION

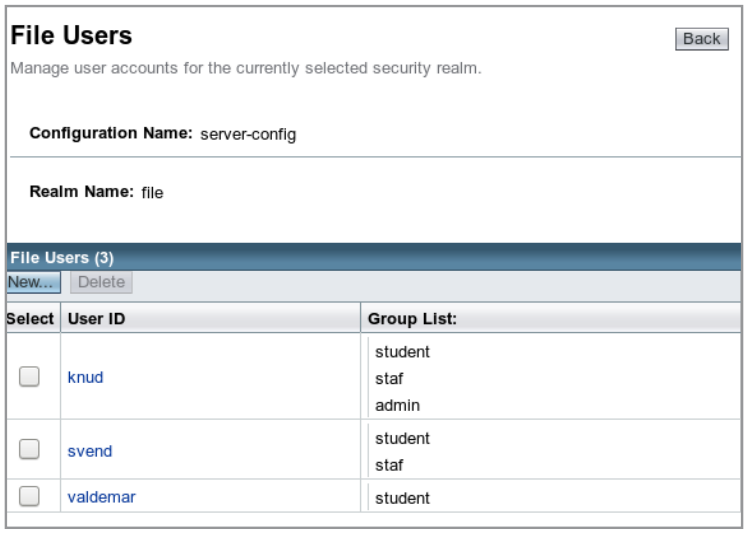
I will start by showing how to let the container (the Glassfish server) control users' access to the individual pages. This is done solely with the configuration files, and without changing the application's code, as well as by creating Glassfish users.

Glassfish manages user and user rights using so-called *security realms*, which basically is a collection of users with associated security groups. A user can be linked to one or more groups, and these groups defines what actions the container will allow the user to perform. For example, an application may have regular users who can only perform specific tasks and administrators who can perform all tasks.

Glassfish has three realms from the start, but you can create your own. The three predefined realms are:

1. *admin-realm*, used solely for the maintenance of users for the Glassfish administration web application and should not be used for users of other applications
2. *certificate,* used for client-side certificate for authentication of users
3. *file*, that stores general user information

Clicking on *file realm* you will get the properties of this realm and clicking the *Manage Users* button will give you an overview of the users. I have created three that I will apply in the following:



I will now use these users for autentication of pages to the above application when

1. users belonging to the *student* group alone must have access to *index.html*
2. users belonging to the group *staf* must have access to *index.xhtml* and *enter.html*
3. users belonging to the *admin* group must have access to all three pages

I have added the following web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/
     javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>

  <security-constraint>
   <web-resource-collection>
    <web-resource-name>Admin Pages</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
    <role-name>admin</role-name>
   </auth-constraint>
  </security-constraint>
```

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>AllPages</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>staf</role-name>
  </auth-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Start Page</web-resource-name>
    <url-pattern>/index.html</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>student</role-name>
  </auth-constraint>
</security-constraint>
```

```
  <login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>file</realm-name>
  </login-config>

</web-app>
```

User rights are defined as *security-constraint* elements, and in this case there are three. A *security-constraint* element defines which pages it concerns and what role the user should have. The first happens with an *url-pattern* element, while the other happens with a *role-name* element. Looking at the first *security-constraint* it applies to all pages where the name starts with *admin*, and thus all pages in the *admin* folder. Only users whose role is *admin* can access these pages. The next *security-constraint* element applies to all pages, but here it is a requirement that the user should have the role *staf*. Finally, there is the last *security-constraint* that applies to a concrete page (start page) and for users whose role is *student*.

The last element is called *login-config* and is used to define how authentication should take place. There are basically three options

1. BASIC, where it is left to the browser to make authentication
2. FORM where it is left for the program to perform authentication
3. CLIENT-CERT, which requires a certificate

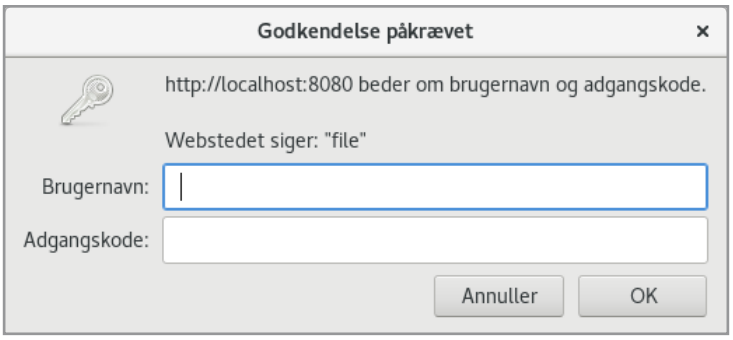
Looking at the above role names, they refer only to the current application, but not to the specific user groups. There must therefore be a mapping of these names, which takes place in *glassfish web.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC " … ">
<glassfish-web-app error-url="">
 <security-role-mapping>
  <role-name>admin</role-name>
  <group-name>admin</group-name>
 </security-role-mapping>
 <security-role-mapping>
  <role-name>staf</role-name>
  <group-name>staf</group-name>
 </security-role-mapping>
 <security-role-mapping>
  <role-name>student</role-name>
  <group-name>student</group-name>
 </security-role-mapping>
 <class-loader delegate="true"/>
 <jsp-config>
```

```
  <property name="keepgenerated" value="true">
    <description>Keep a copy … </description>
  </property>
 </jsp-config>
</glassfish-web-app>
```

In this case, the names are the same, but the meaning is of course that they do not have to be. The server will typically have relatively few groups, while the applications define roles independently of which user groups the server has defined.

If you open the above in Firefox, it will display the following window:



and what you can do with the program depends on which user you log in with.

In this example, *BASIC* authentication is used, which means that the username and password are sent Base64 encoded to the server. It does not provide password protection. The same goes for *FORM* authentication, where the username and password are sent in clear text. Should you solve this problem and you should ensure that all data is sent encrypted between the client and server, then the solution is to use the HTTPS for secure internet protocol, and simply expand *web.xml*:

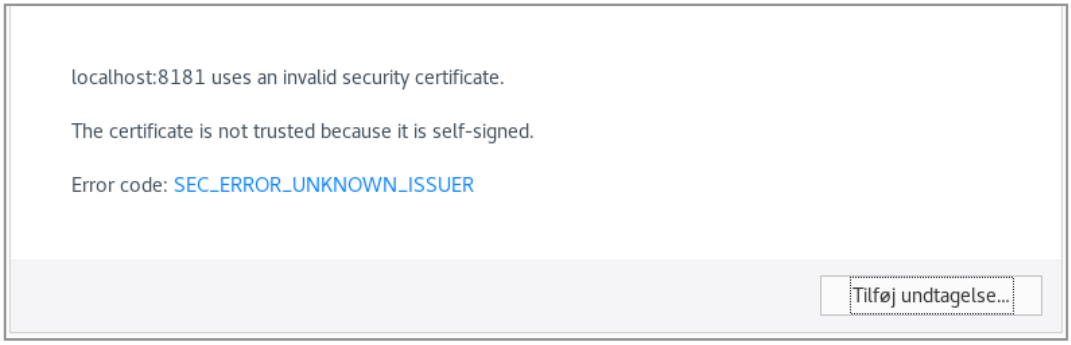
```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http:// … ">
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <security-constraint>
   <web-resource-collection>
    <web-resource-name>Admin Pages</web-resource-name>
```

```
   <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
   <role-name>admin</role-name>
  </auth-constraint>
  <user-data-constraint>
   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
   <web-resource-name>AllPages</web-resource-name>
   <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
   <role-name>staf</role-name>
  </auth-constraint>
  <user-data-constraint>
   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
   <web-resource-name>Start Page</web-resource-name>
```

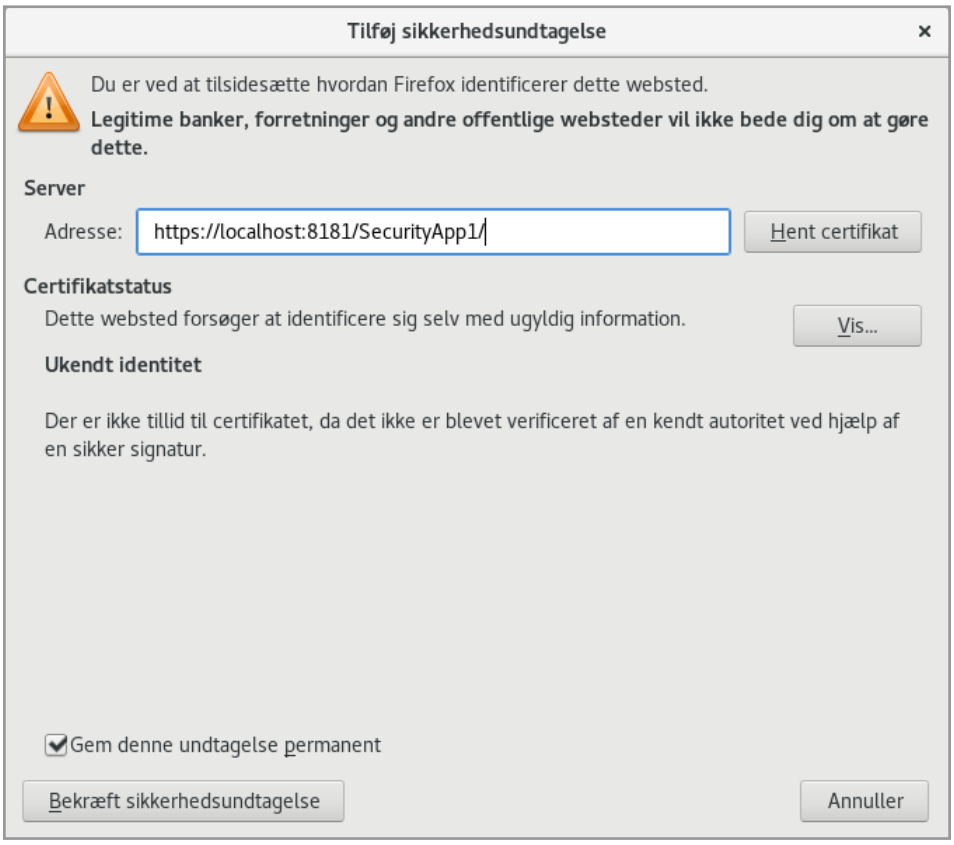
```
    <url-pattern>/index.html</url-pattern>
   </web-resource-collection>
   <auth-constraint>
    <role-name>student</role-name>
   </auth-constraint>
   <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
  </security-constraint>
  <login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>file</realm-name>
  </login-config>
</web-app>
```

Generally, a secure Internet connection requires a certificate that you purchase from a *Certificate Authority* (CA) such as *Verisign* or *Thawte*. Such a certificate costs something (about $500) and needs a renewal every year. It may be a bit unfortunate for development purposes and therefore Glassfish comes with a self-certificate and is pre-configured to support HTTPS at port 8181. Such a self-certificate can be made by everyone and is therefore not sure. If, after the above change of *web.xml*, you open the application, you get the following window (Firefox):

If you click Advanced, you get the following option (which is a warning):



If you click on the button, you'll get another warning:

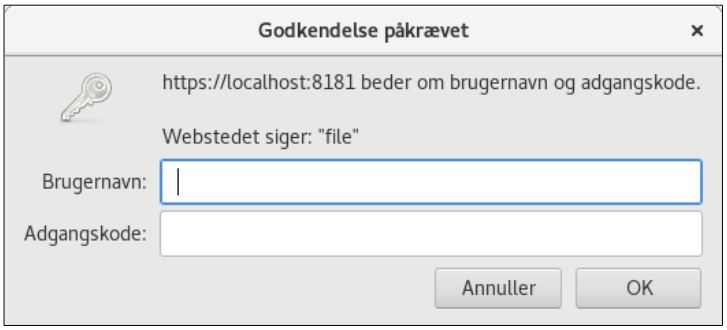


and clicking on the last button means that you have accepted the unsecured certificate and the usual login window appears. If you close the browser and reopen the program, you will not get the above warnings, and the login window will open immediately.

You should note that if you opens the program in the usual way by entering the address

```
http://localhost:8080/SecurityApp1/
```

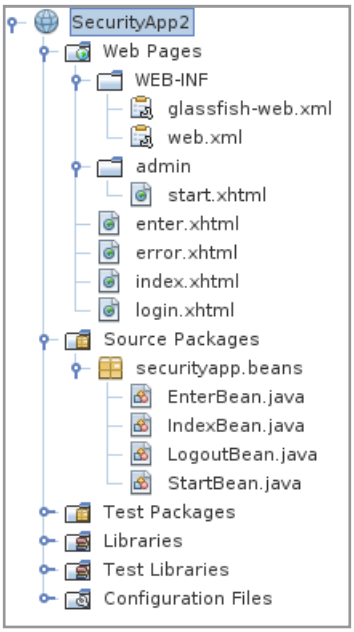
it is the login window as shown below. That is, the address is automatically redirected to the HTTPS protocol and port 8181:



7.3 FORM AUTHENTICATION

In the example above, it is the browser that has shown the login window and it is, in principle, excellent, but you can also write your own login form. There may be at least two reasons for it, including that you decide how the form should look, but it also allows you to implement a logout function. In the following example I will show how to do, but first I will change the program itself.

In the introductory example, the project consists of three HTML pages, but in practice it will typically be JSF pages with backing beans, which requires a few changes. The *SecurityApp2* project is expanded with two new JSF pages compared to the previous project, and in addition, the three original pages have been changed to JSF pages:



The three original pages *index.xhtml*, *enter.xhtml* and *start.xhtml* now have a backing bean. A managed bean with the name *LogoutBean.java* has also been added. The design of the first three pages has changed a bit, but in principle it works in the same way. The *error.xhtml* page is trivial and consists solely of a text and a link to *login.xhtml*, and the page is used if you enter a user that is not found or you have entered an illegal password. The most important thing about the example is the page *login.xhtm*:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC " … ">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  <h2>Enter username and password</h2>
  <form method="POST" action="j_security_check">
   <table cellpadding="0" cellspacing="0" border="0">
    <tr>
     <td align="right">Username: </td>
     <td><input type="text" name="j_username"/></td>
    </tr>
    <tr><td colspan="2" style="height:10px"></td></tr>
    <tr>
     <td align="right">Password: </td>
     <td><input type="password" name="j_password"/></td>
    </tr>
    <tr><td colspan="2" style="height:20px"></td></tr>
```

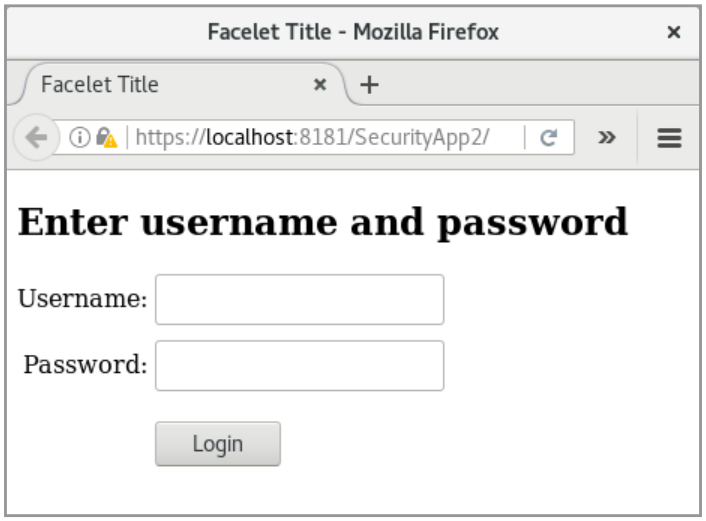
```
     <tr>
       <td></td>
       <td><input type="submit" value="Login"/></td>
     </tr>
    </table>
   </form>
  </h:body>
</html>
```

It is a JSF page, but no JSF elements are used, and in particular it is important that the *form* element is not a *h:form* element. Here you should note the *action* attribute, which refers to *j_security_check*, which specifies the code that the application server performs. For the same reasons, the two input fields must be named *j_username* and *j_password*.

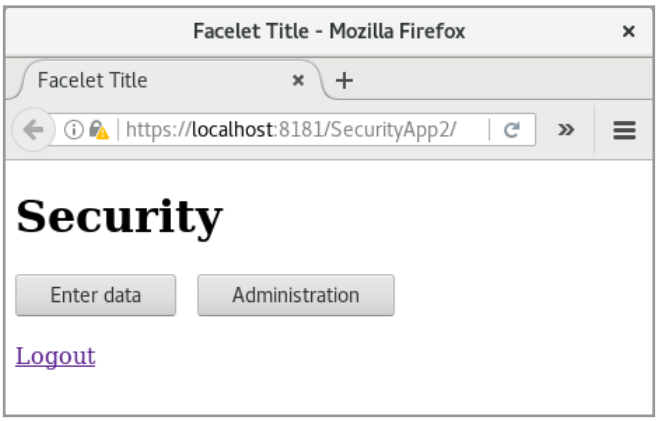
The last thing that must happen to use form authentication is the change of *web.xml*, where the *login-config* element is changed to the following:

```
<login-config>
 <auth-method>FORM</auth-method>
 <realm-name>file</realm-name>
 <form-login-config>
  <form-login-page>/faces/login.xhtml</form-login-page>
  <form-error-page>/faces/error.xhtml</form-error-page>
 </form-login-config>
</login-config>
```

When you opens the application, you will see the window below, where you will enter the username and password. In this case, it is a very simple form, but you can of course modify *login.xhtml*, so you get a nicer design that fits the current application. Note that the address bar shows that a secure internet connection is being used.



After entering your username and password, you will get to *index.xhtml*:



The window looks a bit different, but it is important that there is a link to logout. As mentioned above, the page has a managed bean:

```
package securityapp.beans;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
```

```
@Named(value = "indexBean")
@SessionScoped
public class IndexBean implements Serializable
{
 public String admin()
 {
  return "admin/start?faces-redirect=true";
 }

 public String enter()
 {
  return "enter?faces-redirect=true";
 }
}
```

The two action methods should do nothing but redirect to the two pages *start.xhtml* and *enter.xhtml* that happens when you click on one of the two buttons. When a JSF page performs an action, it happens with the POST method, and instead of performing a redirection, a forward is performed. Therefore, the user role is not validated (or it only happens when you leave a page again). To solve the problem (so the application works like the previous application), as described above, you must force a redirection.

Clicking on the *Logout* link, is called a method *logout()* in a manged bean *LogoutBean.java*:

```
package securityapp.beans;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import javax.faces.context.*;
import javax.servlet.http.*;

@Named(value = "logoutBean")
@RequestScoped
public class LogoutBean
{
 public LogoutBean()
 {
 }

 public String logout()
 {
  ExternalContext externalContext =
    FacesContext.getCurrentInstance().getExternalContext();
  HttpSession session = (HttpSession) externalContext.getSession(true);
  session.invalidate();
  return "login?faces-redirect=true";
 }
}
```

The method determines a reference to the current session and invalidates it (all session objects are removed). Then the user is redirected to the login window.

7.4 CLIENT CERTIFICATE

Instead of making authentication using a username and password, you can use a certificate that you can purchase from a *Certificate Authority* (*Verisign* or *Thawte*) as mentioned above. This certificate must be installed by both Glassfish and the browser. For testing and development purposes, it is possible to create a self-signed certificate. This is done with a tool called *keytoll*, which comes with jdk. On my machine, the *keytoll* is placed in

```
/usr/java/jdk1.8.0_131/bin
```

but it depends, of course, on where Java is installed and what version. First, I'll create a *keystore* with a key pair, which can be done with the following command, where the user through a simple dialog must enter values that can identify the user and his company (these are, among other things, those values that a Certificate Authority verifies):

```
./keytool -genkey -v -alias pakey -keyalg RSA -storetype PKCS12 -keystore
client_keystore.p12 -storepass Volmer1234 -keypass Volmer1234
What is your first and last name?
 [Unknown]: Poul Klausen
What is the name of your organizational unit?
 [Unknown]: Development
What is the name of your organization?
 [Unknown]: Torus Data
What is the name of your City or Locality?
 [Unknown]: Skive
What is the name of your State or Province?
 [Unknown]: Jylland
What is the two-letter country code for this unit?
 [Unknown]: dk
Is CN=Poul Klausen, OU=Development, O=Torus Data, L=Skive, ST=Jylland, C=dk
correct?
 [no]: yes
Generating 2.048 bit RSA key pair and self-signed certificate (SHA256withRSA)
with a validity of 90 days
   for: CN=Poul Klausen, OU=Development, O=Torus
Data, L=Skive, ST=Jylland, C=dk
[Storing client_keystore.p12]
```

For most information, it is not important (for a self-signed certificate) what you write, but for the arguments of the command, the name of your keystore must end with *.p12* and the *storepass* and the *keypass* should be the same. After the command has been completed, a keystore named *client_keystore.p12* has been created under your current directory. Your

keystore must be imported into the browser, what you do (in Firefox), by choose *Settings* and *Advanced*, and here you must choose *Certificates* and again *Show Certificates*:

In the following window you will get a list of the certificates that are installed in your browser and clicking on the *Import* button, you can browse to your keystore (the file *client_keystore.p12*) and install it.

The certificate must be exported to Glassfish, and first it must be converted to a format that Glassfish can use. This can again be done with *keytool*:

```
./keytool -export -alias pakey -keystore client_keystore.p12
-storetype PKCS12 -storepass Volmer1234 -rfc -file pasigned.cer
Certificate stored in file <pasigned.cer>
```

The result is a file named *pasigned.cer*. The name is not important, but it is recommended that the name end with *.cer*. Glassfish has a file with the names of the *Certification Authorities* that it trusts (it already knows *Verisign* and *Thaute*), and these names are found in the file

```
glassfish/domains/domain1/config/cacerts.jks
```

In order for Glassfish to use my self-signed certificate, it must know me – and believe me as a *Certificate Authority* – and I can export the file *pasigned.cer* to Glassfish with keytool:

```
./keytool -import -file pasigned.cer -keystore /usr/
local/glassfish-4.1.1/glassfish/domains/domain1/config/
cacerts.jks -keypass changeit -storepass changeit
Owner: CN=Poul Klausen, OU=Development, O=Torus
Data, L=Skive, ST=Jylland, C=dk
Issuer: CN=Poul Klausen, OU=Development, O=Torus
Data, L=Skive, ST=Jylland, C=dk
Serial number: 17f52e72
Valid from: Fri Nov 03 14:32:02 CET 2017
until: Thu Feb 01 14:32:02 CET 2018
Certificate fingerprints:
   MD5: 7C:FD:D5:54:A3:77:91:FE:37:1B:C9:87:0E:7C:31:32
   SHA1: CA:04:02:46:99:B8:3A:0C:E5:04:FA:7D:97:42:39:60:51:27:01:66
   SHA256: 85:F4:6B:07:1E:B5:B1:F9:BE:45:D3:64:9D:43:E4:
83:23:FE:7B:CD:22:C3:5E:42:E9:80:72:61:43:4D:F1:1B
   Signature algorithm name: SHA256withRSA
   Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: A6 FC A4 3C B7 FA 9E 17 25 BD FE FE 48 9B 7E D2 …<….%…H…
0010: 0F D2 04 97 ….
]
]
Trust this certificate? [no]: yes
Certificate was added to keystore
```

Then I can use the certificate for authentication. I want to use a copy of the *SecurityApp1* project, which I have called *SecurityApp3*. The code should not be changed, but the *web.xml* and *glassfish-web.xml* configuration files should. Below is *web.xml* shown:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=" … >
 <session-config>
   <session-timeout>
     30
   </session-timeout>
 </session-config>

 <security-constraint>
  <web-resource-collection>
   <web-resource-name>AllPages</web-resource-name>
   <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
   <role-name>student</role-name>
  </auth-constraint>
  <user-data-constraint>
   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
 </security-constraint>

 <login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>certificate</realm-name>
 </login-config>
```

This time, there is only one single security-constrant element, since all users with the role *student* must have access to the entire application and also defines that a secure Internet connection should be used. Additionally, you can not use authentication with a certificate without an SSL connection. Then there is the *login-config* element, which defines the use of a certificate this time.

Which certificate to use is defined in *glassfish-web.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD
```

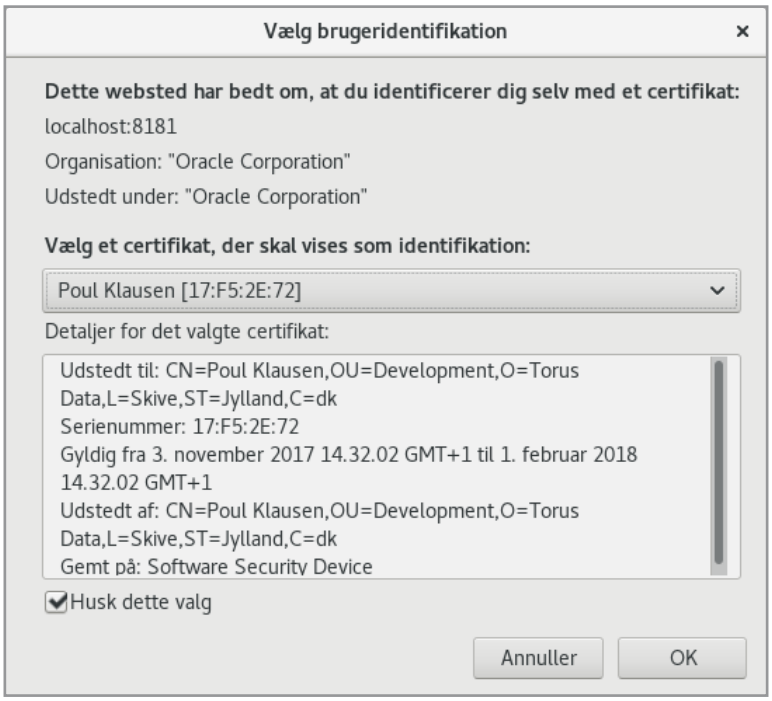
```
GlassFish Application Server 3.1 Servlet 3.0//EN" "http://
glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app error-url="">

<context-root>/certificaterealm</context-root>
 <security-role-mapping>
  <role-name>student</role-name>
  <principal-name>
   CN=Poul Klausen, OU=Development, O=Torus Data, L=Skive, ST=Jylland, C=dk
  </principal-name>
 </security-role-mapping>
 <class-loader delegate="true"/>
 <jsp-config>
  <property name="keepgenerated" value="true">
   <description>Keep a copy of the generated … </description>
  </property>
 </jsp-config>
</glassfish-web-app>
```

Here the user role *student* must be mapped to the certificate. Here it is important that the *principal-name* element is written correctly. It is recommended to use *keytool*:

```
 ./keytool -printcert -file pasigned.cer
Owner: CN=Poul Klausen, OU=Development, O=Torus
Data, L=Skive, ST=Jylland, C=dk
Issuer: CN=Poul Klausen, OU=Development, O=Torus
Data, L=Skive, ST=Jylland, C=dk
…
```

and then copy from the *Issuer* line.

After you have translated and deployed the application and opens it in the browser, you get the following window:



and clicking OK will open the application as before, except that all pages are now accessible.

7.5 PROGRAMMER DEFINED AUTHENTICATION

In the above examples, it is the application server that has handled authentication, and in many cases it is also the way to do it, but the method is best to control the application's authentication and thus whether a user has access to a particular application. On the other hand, if you want to ensure that all users do not have access to the same features (pages) in a web application, you need a bit more, and you can use a database that contains roles for the individual users and then in the application test which roles are available. In this section I will illustrate it with a copy of the example *SecurityApp2*, which I have called *SecurityApp4*.

Basically, it is the same application with the same pages *index.xhtml*, *enter.xhtml* and *start.xhtml* (the last in the folder *admin*) and they have the same backing beans. However, another trivial page has been added named *list.xhtml* and its associated backing bean, to which there is a link from *index.xhtml*. The whole application is thus trivial so far, and the sole purpose is to show how to implement authentication, and in addition to how to make access to each page depending on what it is for a user that is logged in.

Initially, I have created a database called *authentication* and with a single table *user*:

```
use sys;

drop database if exists authentication;
create database authentication;

use authentication;

create table user (
 name varchar(256) not null,
 role varchar(2048),
 primary key (name)
);

insert into user (name, role) values ('poul', 'student:root');
insert into user (name, role) values ('knud', 'admin:staf:student');
insert into user (name, role) values ('svend', 'staf:student');
insert into user (name, role) values ('valdemar', 'student');
```

The table has only two columns, where the first represents the name of a user and is the primary key, while the second column indicates user roles according to the same syntax as in Glassfish. As examples, 4 users are added to the table.

In the project, I have deleted *LogoutBean* and instead added another named bean with the name *AuthenticationCtrl*, but before I look at it, I will show an EJB that contains the most important to login. It is still the application server that needs to authenticate and thus check the user's username and password. Note that the above database table does not contain the user's password, and it is thus the server who is responsible for storing the password encrypted.

To write the EJB I have created an EJB Module project named *AuthenticationEJB*. Then here I created a class library project named *AuthenticationRemote* and added an entity class for the above table. This class has not been changed in relation to what NetBeans has generated and the content is not new and will not appear here. For *AuthenticationEJB*, I've added a reference

to the above class library and added a stateless session bean named *AuthenticationBean* with a remote interface located in the class library. The interface is as follows:

```
package authenticationejb.beans;

import javax.ejb.Remote;
import authenticationremote.entities.User;

@Remote
public interface AuthenticationBeanRemote
{
 public String getPassword();
 public void setPassword(String password);
 public User getUser();
 public void setUser(User user);
 public boolean login();
}
```

which defines two properties, one being the user's password (*AuthenticationBean* must use it to perform authentication using Glassfish), and an object of the type *User*, which is the entity type from the class library. Finally, there is the method *login()*, which is the method of authentication. Below is the class *AuthenticationBean*:

```
@Stateless
public class AuthenticationBean implements AuthenticationBeanRemote
{
 @PersistenceContext(unitName = "AuthenticationEJBPU")
 private EntityManager em;
 private String password = null;
 private User user;

 public AuthenticationBean()
 {
 }

 @Override
 public boolean login()
 {
  HttpSession session = getSession();
  HttpServletRequest request = null;
  try
  {
   request = (HttpServletRequest)
    FacesContext.getCurrentInstance().getExternalContext().getRequest();
   request.login(getUser().getName(), this.password);
   session.setMaxInactiveInterval(600);
   session.setAttribute("authenticated", new Boolean(true));
   em.flush();
   Query query =
    em.createQuery("select count(u) from User u where u.name = :name").
     setParameter("name", getUser().getName());
   Long count = (Long)query.getSingleResult();
   if (count > 0)
   {
    query =
     em.createQuery("select object(u) from User u where u.name = :name").
      setParameter("name", getUser().getName());
    setUser((User) query.getSingleResult());
   }
   return true;
  }
  catch (Exception ex)
  {
   setUser(null);
   session = getSession();
   session.setAttribute("authenticated", new Boolean(false));
   if(request != null)
   {
    try
    {
     request.logout();
```

```
    }
    catch (Exception e)
    {
    }
   }
   return false;
  }
  finally
  {
   setPassword(null);
  }
 }

 public HttpSession getSession()
 {
  HttpServletRequest request = (HttpServletRequest)
   FacesContext.getCurrentInstance().getExternalContext().getRequest();
  return request.getSession(false);
 }

 @Override
 public String getPassword()
 {
  return this.password;
 }

 @Override
 public void setPassword(String password)
 {
  this.password = password;
 }

 @Override
 public User getUser()
 {
  if (this.user == null) user = new User();
  return user;
 }

 @Override
 public void setUser(User user)
 {
  this.user = user;
 }
}
```

The class implements the two properties, which do not require any particular explanation, and the important thing is naturally the method *login()*. The first thing that happens is that the method initializes the variable *request*, which represents the current user request. For this request, the method *login()* is called with the user's username and password as parameters (provided they have a value at that time) and that means authentication using the server. If the server can not autheticate the user you get an exception, but otherwise the timeout period for the current session is set to 10 minutes, and a session object is defined with the value *true*, which indicates that login has been made. Next, it is checked whether the user is in the database, and if that is the case, the propertien *user* is initialized. If the server can not perform authentication, a *logout()* method will be performed on *request*, which means that you will be returned to the login page.

Then there is *AuthenticationCtrl*, which is a manged bean. It uses *AuthenticationBean* and generally represents two properties *username* and *password* for a login form. In addition, there is a property *authenticated* that indicates whether a user is logged in.

```
@Named(value = "authenticationCtrl")
@SessionScoped
public class AuthenticationCtrl implements Serializable
{
```

```
@EJB
private AuthenticationBeanRemote authenticationBean;
private User user;
private boolean authenticated = false;

public AuthenticationCtrl()
{
}

public String getUsername()
{
 return getUser().getName();
}

public void setUsername(String username)
{
 getUser().setName(username);
}

public String getPassword()
{
 return authenticationBean.getPassword();
}

public void setPassword(String password)
{
 authenticationBean.setPassword(password);
}

public User getUser()
{
 if (this.user == null)
 {
  user = new User();
  setUser(authenticationBean.getUser());
 }
 return user;
}

public void setUser(User user)
{
 this.user = user;
}

public boolean isAuthenticated()
{
 try
 {
```

```
  authenticated = (Boolean) getSession().getAttribute("authenticated");
 }
 catch (Exception ex)
 {
  authenticated = false;
 }
 return authenticated;
}

public void setAuthenticated(boolean authenticated)
{
 this.authenticated = authenticated;
}

public boolean isStaff()
{
 return user != null && hasRole(user.getRole(), "staf");
}

public boolean isAdmin()
{
 return user != null && hasRole(user.getRole(), "admin");
}

public String login()
{
 authenticationBean.setUser(getUser());
 boolean authResult = authenticationBean.login();
 if (authResult)
 {
  authenticated = true;
  setUser(authenticationBean.getUser());
  return "index";
 }
 else
 {
  authenticated = false;
  setUser(null);
  return "";
 }
}

public String logout()
{
 user = null;
 this.authenticated = false;
 authenticationBean.setUser(null);
 ExternalContext externalContext =
  FacesContext.getCurrentInstance().getExternalContext();
```

```
  externalContext.invalidateSession();
  return "login?faces-redirect=true";
 }

 public HttpSession getSession()
 {
  HttpServletRequest request = (HttpServletRequest)
    FacesContext.getCurrentInstance().getExternalContext().getRequest();
  return request.getSession();
 }

 private boolean hasRole(String roles, String role)
 {
  String[] elems = roles.split(":");
  for (String elem : elems) if (elem.equals(role)) return true;
  return false;
 }
}
```

There are also two properties *isAdmin()* and *isStaff()* that are used to test the roles of those users. They are used by *index.xhtml* to make the content (if the buttons are displayed) depending on the current user's roles:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC " … >
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  <h1>Security</h1>
  <h:form>
   <h:commandLink value="Your names" action="#{indexBean.show()}" /><br/><br/>
   <h:commandButton rendered="#{authenticationCtrl.staff}" value="Enter data"
    action="#{indexBean.enter()}" />  
   <h:commandButton rendered="#{authenticationCtrl.admin}"
    value="Administration" action="#{indexBean.admin()}" /><br/><br/>
   <h:commandLink value="Logout" action="#{authenticationCtrl.logout()}" />
  </h:form>
 </h:body>
</html>
```

Finally, the page *login.xhtml* has been changed to a regular JSF page:

```
<h:body>
 <h2>Enter username and password</h2>
 <h:form id="login">
  <h:panelGroup rendered="#{authenticationCtrl.authenticated}">
   <a href="index.xhtml">Authenticated successfully…go to Application</a>
  </h:panelGroup>
  <h:panelGrid columns="2" rendered="#{!authenticationCtrl.authenticated}" >
   <h:outputText value="Username:" />
   <h:inputText id="j_username" value="#{authenticationCtrl.username}"/>
   <h:outputText value="Password:" />
   <h:inputText id="j_password" value="#{authenticationCtrl.password}"/>
   <h:panelGroup/>
   <h:commandButton id="login" action="#{authenticationCtrl.login}"
    value="Login"/>
  </h:panelGrid>
 </h:form>
</h:body>
```

With the above solution, the application server is responsible for authentication and encryption of the users passwords while it is the program and thus the programmer that is responsible for which pages each users can access. Of course it is also possible to take care of everything and let it be up to the program to handle everything about users. Here you should be aware that the users passwords are stored encrypted in the database.

8 A FINAL EXAMPLE

As the final example, I will write a program that is a typical calculator, and thus a program that can simulate a mathematical calculator. Compared to previous examples (the book Java 3), there must be two important differences:

1. The program must now be a web application.
2. The machine (the calculator) must now support calculations with a random number of decimal places.

Apart from that, the program should basically have the same functions as a typical (and relatively simple) mathematical calculator, where it is important that the program should support very large numbers.

The aim of the program is to use some of the topics that have been addressed in this book, and especially Enterprise Java Beans. In addition, it is an example where there is focus on algorithms.

8.1 ANALYSIS

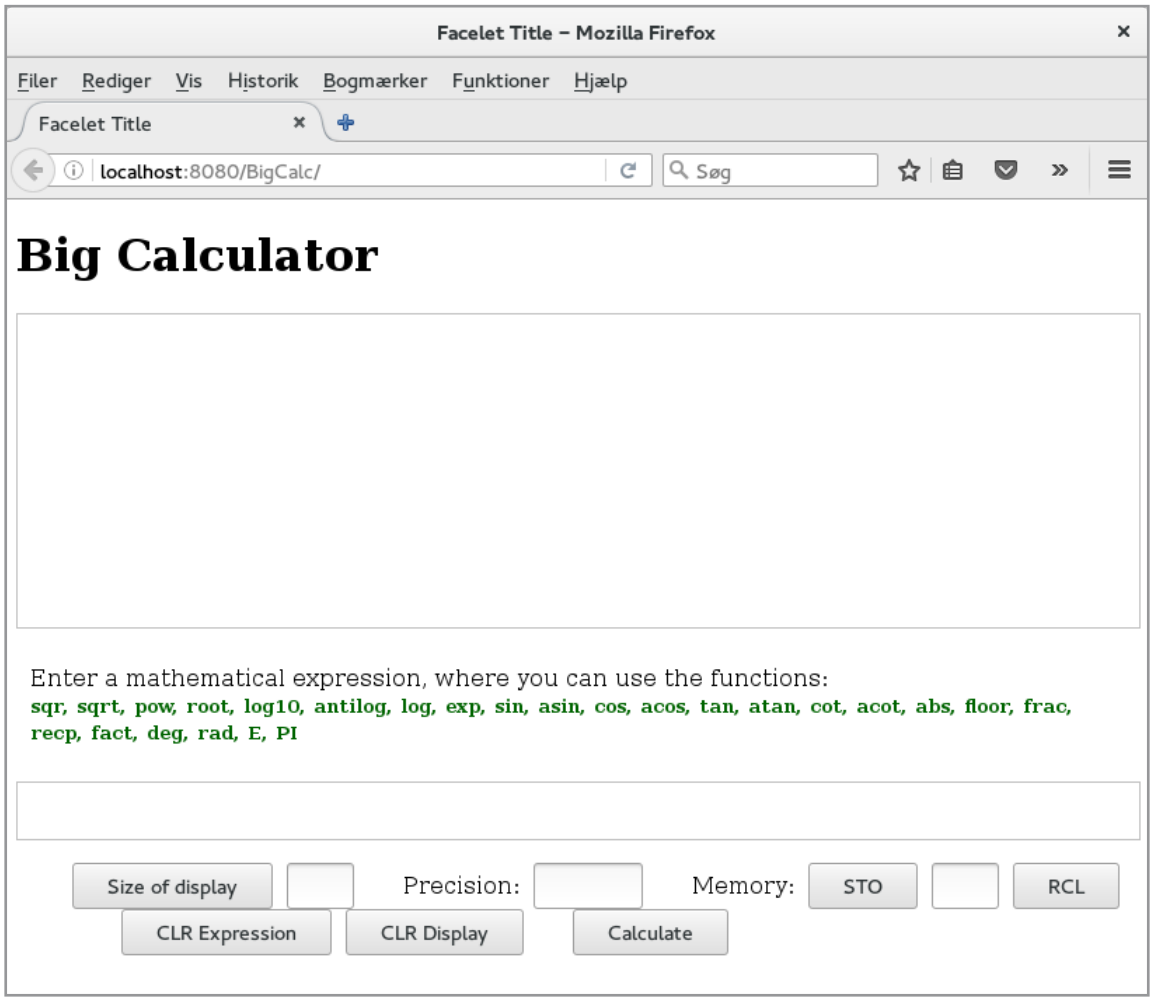
Regarding the last of the two requirements, it means that the program must work on data of the type *BigDecimal*.

The program must support the following mathematical functions:

1. *sqr(x)* which calculates the square of *x*, it is x^2
2. *sqrt(x)* which calculates the square root *x*, it is $\sqrt{(x)}$
3. *pow(x,y)* which calculates x^y
4. *root(x, y)* which calculates the y'th root of *x*, it is $\sqrt[y]{x}$
5. *log10(x)* which calculates the 10th logarithm of *x*
6. *antilog(x)* which calculates the anti logarithm of *x* and then 10^x
7. *log(x)* which calculates the natural logarithm to *x*
8. *exp(x)* which is the eksponential function and then calculates e^x
9. *sin(x)* which calculates sinus to *x*
10. *asin(x)* which is the reverse function for *sinus*
11. *cos(x)* which calculates *cosines* to *x*
12. *acos(x)* which is the reverse function for *cos*
13. *tan(x)* which calculates *tanges* to *x*

14. *atan(x)* which is the reverse function for *tan*
15. *cot(x)* which calculates *cotangens* to *x*
16. *acot(x)* which is the reverse function for *cotangens*
17. *abs(x)* which calculates the absolut value of *x*
18. *floor(x)* which calculates the integer value of *x*
19. *frac(x)* which calculates the fraction of *x*
20. *recp(x)* which calculates the reciproc value of *x*
21. *fact(x)* which calculates the factorial of *x*
22. *deg(x)* which calculates the value of *x* in radians to degrees
23. *rad(x)* which calculates the value of *x* in degrees to radians
24. *E* which is the constant *e*
25. *PI* which is the constant π

The program should have a user interface similar to the following:



At the top there is a display for results. The next field is an entry field for a mathematical expression. The last line of commands indicates how many lines should be in the display, which precision should be used to calculate expressions, commands to save and retrieve values from a registry where you enter the name, two buttons to clear the above fields and finally a button to perform a calculation.

The program is thus, in principle, a rather simple program and the biggest challenge is to implement the mathematical algorithms. It can be a problem to implement them with sufficient efficiency.

8.2 DESIGN

During the design I will look at the design of the algorithms for the mathematical functions as well as the overall program architecture.

Design of algorithms

I want to start with the algorithms that can be used to implement the mathematical functions. Most are simple, others are standard, while for others you can find good solutions on the Internet. The arithmetic itself is implemented using the class *java.math.BigDecimal*. It is not difficult to find algorithms for the functions mentioned in the analysis, but the problem is to produce effective algorithms, as many of the following formulas become ineffective along the asymptotes. It may therefore be necessary to implement special solutions for values that are asymptotic.

public BigDecimal sqr(BigDecimal x)

$sqr: \mathbb{R} \to \mathbb{R}_0$

The function is defined on the entire real axis and is bijective on the non-negative half line. The algorithm is trivial, as it is simply a simple multiplication.

public BigDecimal sqrt(BigDecimal x) throws BigException

$\text{sqrt}: \mathbb{R}_0 \to \mathbb{R}_0$

The function is bijective, and the reverse is the restriction of *sqr*. Often, the function is implemented using the so-called *Babylonian* method (but there are other options) based on the following sequence:

$$x_0 \approx \sqrt{(x)}$$
$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{x}{x_n}\right)$$
$$\lim_{x \to \infty} x_n = \sqrt{(x)}$$

The algorithm is characterized by that it fast convergence, but it depends on how "lucky" one is with the first estimate (x_0), but even if you choose $x_0 = x$ the method is nevertheless effective.

The method raises an exception if the argument is negative.

public BigDecimal pow(BigDecimal x, BigDecimal y) throws Exception

$pow: \mathbb{R}_0 \to \mathbb{R}_0$

and applies to any *y* different from 0. Is y = 0, it is the constant function with the value 1. The algorithm is simple and the value can be determined as $exp(ylog(x))$. If x is not positive, an exception is raised.

public BigDecimal root(BigDecimal x, BigDecimal y) throws Exception

Is generally the same algorithm as above, and the value can be determined as $exp\left(\frac{1}{y}log(x)\right)$. It is required that y is not 0. Otherwise an exception is raised.

public BigDecimal log10(BigDecimal x) throws BigException

$log10: \mathbb{R}_+ \rightarrow \mathbb{R}$

is bijective and the reverse is *antilog*. There are several excellent algorithms based on the Taylor series, which can be used to determine the logarithm of a number, but they all have in common that they become ineffective when x approaches the asymptotes. When the argument is represented as a *BigDecimal*, and as it is the logarithm with base number 10, the algorithm can proceed as follows:

```
d = the number of digets in front of the decimal point in x-1
add d to the result
add '.' to the result
loop over the desired number of decimals
{
 move the decimal point in x d places to the left
 raise x in the power og 10
 d = the number of digets in front of the decimal point in x-1
 add d to the result
}
```

The algorithm only applies to x if it is greater than or equal to 1. Is x less than 1, you can use the same algorithm, but on $\frac{1}{x}$ and use that $log_{10}\left(\frac{1}{x}\right) = -log_{10}(x)$.

The method raises an exception if x is negative or 0.

public BigDecimal antilog(BigDecimal x) throws BigException

$antilog: \mathbb{R} \rightarrow \mathbb{R}_+$

The function can be implemented using *exp* and *log*: exp(*x* log(10))

public BigDecimal log(BigDecimal x) throws BigException

With *log10* available, the method is trivial as it is just log10 multiplied by a constant:

$$log(x) = \frac{log10(x)}{log10(e)}$$

where *e* is Euler's constant.

public BigDecimal exp(BigDecimal x) throws BigException

$esp: \mathbb{R} \rightarrow \mathbb{R}_+$

The method can be implemented relatively efficiently using the Taylor series:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

public BigDecimal sin(BigDecimal x)

$sin: \mathbb{R} \rightarrow [-1; 1]$

The function is periodic with the period 2π. The function can be implemented using the Taylor series:

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

public BigDecimal asin(BigDecimal x) throws BigException

The restriction of sinus:

$$sin: \left[\frac{-\pi}{2}; \frac{\pi}{2}\right] \to [-1; 1]$$

is a bijection and the reverse function is

$$asin: [-1; 1] \to \left[\frac{-\pi}{2}; \frac{\pi}{2}\right]$$

and can be defined using atan:

$$asin(x) = 2atan\left(\frac{x}{1 + \sqrt{(1 - x^2)}}\right)$$

public BigDecimal cos(BigDecimal x)

$$cos: \mathbb{R} \to [-1; 1]$$

The function is periodic with the period 2π. The function can be implemented using the Taylor series:

$$cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

public BigDecimal acos(BigDecimal x) throws BigException

The restriction of cosinus:

$$cos: [0; \pi] \to [-1; 1]$$

is a bijection and the reverse function is

$$acos: [-1; 1] \to [0; \pi]$$

and can be defined using atan:

$$acos(x) = 2atan\left(\frac{\sqrt{(1-x^2)}}{1+x}\right)$$

However, the formula applies only to $x > -1$.

public BigDecimal tan(BigDecimal x) throws BigException

tan is defined for all real numbers x, where cos(x) is not 0, that is, all numbers that are not of the form $\frac{3p\pi}{2}$. The implementation is trivial:

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

public BigDecimal atan(BigDecimal x) throws BigException

If you consider the restriction of tan:

$$tan:]-\frac{\pi}{2}; \frac{\pi}{2}[\rightarrow]-1; 1[$$

is it a bijection and thus has a revere

$$atan:]-1;1[\rightarrow]-\frac{\pi}{2};\frac{\pi}{2}[$$

The function has a Taylor series, but it is not very effective. There are several alternatives, and one example is:

$$atan(x) = \sum_{n=0}^{\infty} \frac{2^{2n}(n!)^2 x^{2n+1}}{(2n+1)!\,(1+x^2)^{n+1}}$$

This algorithm is relatively effective as long as x is not much larger than 1. If so, one can apply

$$atan(x) = \frac{\pi}{2} - atan\left(\frac{1}{x}\right)$$

This implementation of *atan* is reasonable – but important, since other functions depend on it.

public BigDecimal cot(BigDecimal x) throws BigException

cot is defined for all real numbers *x*, where *sin(x)* is not 0, that is, all numbers that are not og the form $\frac{p\pi}{2}$. The implementation is er trivial:

$$cot(x) = \frac{cos(x)}{sin(x)}$$

public BigDecimal acot(BigDecimal x) throws BigException

Considering the restriction of *cot*:

$$\cot:]0;\pi[\rightarrow]-1;1[$$

is it a bijection and thus has a reverse

$$acot:]-1;1[\rightarrow]0;\pi[$$

The function can be defined as

$$acot(x) = \frac{\pi}{2} - atan(x)$$

public BigDecimal abs(BigDecimal x)

This feature is trivial as it is directly supported by the class *BigDecimal*.

public BigDecimal floor(BigDecimal x)

Also a simple method as it can be implemented using the method *round()* of *BigDecimal*.

public BigDecimal frac(BigDecimal x)

Can be implemented using the method *floor()* and subtraction.

public BigDecimal recp(BigDecimal x) throws BigException

The method is nothing but a division, but can raise an exeption by division with 0.

public BigDecimal fact(BigDecimal x) throws BigException

It is one of the complex methods that are not actually implemented optimally. The argument is a real number and the faculty of a real number is set to 1 if the argument x is less than or equal to 0. Is the argument positive the factorial is defined as

$$x! = \Gamma(x+1)$$

where Γ is the *gamma* function. The problem is that it is difficult to implement the gamma function with satisfactory efficiency. Typically, one use

$$\Gamma(x) = exp\left(log\big(\Gamma(x)\big)\right)$$

Because it is relatively simple to implement $exp(x)$ effectively, the problem is reduced to implement $log\big(\Gamma(x)\big)$. It is also not simple, but there are several formulas that are an approximation, and one of them is:

$$log\big(\Gamma(x)\big) = \frac{1}{2}log(2\pi) + \left(x - \frac{1}{2}\right)log(x) - x - \frac{x}{2}log\left(xsinh\left(\frac{1}{x}\right) + \frac{1}{810x^6}\right)$$

See for example:

`https://math.stackexchange.com/questions/19236/`
`algorithm-to-compute-gamma-function)`

This function is, in principle, simple to implement, but requires that you also implement *sinh* (see below).

The above formula determines an approximated value, which is not good if the argument is an integer. Therefore, the method *fact()* must handle the special case where the argument can be perceived as an integer, so that the faculty is only determined by multiplication.

public BigDecimal deg(BigDecimal x)

This function is also simple as the conversion from radians to degrees can be done with the expression

$$xradianer = 180\frac{x}{\pi}.$$

public BigDecimal rad(BigDecimal x)

This function is similar to where $xdegrees = \frac{x\pi}{180}$.

PI og E

Finally, there are the two constant functions *PI* and *E*. These functions are hardcoded from files containing the two constants with 1000000 significant digits.

For the sake of implementing the function *log(gamma(x))*, there is also a need to implement *sinh*, and for completeness, I will implement all four hyperbolic functions as well as their reverses.

public BigDecimal sinh(BigDecimal x) throws BigException

$$sinh: \mathbb{R} \to \mathbb{R}$$

is a bijection. There is a formula expressed by the exponential function:

$$sinh(x) = \frac{e^x - e^{-x}}{2}$$

public BigDecimal asinh(BigDecimal x) throws BigException

The inverse of *sinh* can be determined by the following formula:

$$asinh(x) = log\left(x + \sqrt{(x^2 + 1)}\right)$$

public BigDecimal cosh(BigDecimal x) throws BigException

$$\cosh: \mathbb{R} \to [1; \infty[$$

There is a formula expressed by the exponential function:

$$cosh(x) = \frac{e^x + e^{-x}}{2}$$

public BigDecimal acosh(BigDecimal x) throws BigException

The restriction of *cosh* to $[0; \infty[$ is a bijection and the reverse to *cosh* can be calculated from the formula:

$$acosh: [1; \infty[\to [0; \infty[\; acosh(x) = log\left(x + \sqrt{(x^2 - 1)}\right)$$

public BigDecimal tanh(BigDecimal x) throws BigException

$$tanh: \mathbb{R} \to]-1; 1[$$

The function is bijective and is defined as:

$$tanh(x) = \frac{sinh(x)}{cosh(x)}$$

public BigDecimal atanh(BigDecimal x) throws BigException

$$atanh:]-1;1[\rightarrow \mathbb{R}$$

$$atanh(x) = \frac{1}{2} log\left(\frac{1+x}{1-x}\right)$$

public BigDecimal coth(BigDecimal x) throws BigException

$$coth: \mathbb{R} - \{0\} \rightarrow \mathbb{R} - \{0\}$$

The function is defined as:

$$coth(x) = \frac{cosh(x)}{sinh(x)}$$

public BigDecimal acoth(BigDecimal x) throws BigException

The restriction of *coth* defines a bijection

that can be calculated from the formula:

$$acot(x) = \frac{1}{2} log\left(\frac{x+1}{x-1}\right)$$

Program design

The program should be a simple web application with a single page. Since the above functions may be used by other programs, it has been decided to implement a class library that contains the functions. The web application should use a stateless session bean that uses the class library and this EJB must provide two services:

BigDecimal evaluate(String expression)

which will evaluate the value of a math expression and return the result as a *BigDecimal*. In addition, there must be a service

List<String> reserved()

which returns all the function names (reserved words) that the EJB supports.

The method of *evaluate()* is an extremely complex method, since the argument is a mathematical expression entered as a string, and therefore the method must both scan, parse and evaluate the expression. However it is the same task that I have previously looked at in the final example of the book Java 3. The same algorithm/solution can therefore be used because the difference is primarily that the data type in Java 3 was *double* while in this example it is *BigDecimal*.

Then the solution of the task will include the following NetBean projects:

1. *FunctionsLib*, which is a Class Library project that implements the mathematical functions.
2. *FunctionsRemote*, which is a Class Library project that is a remote interface for the EJB.
3. *FunctionsEJB*, which is an EJB module with a single stateless session bean.
4. *BigCalc*, which is the web application.

In addition, there will be a usual Java Application project *BigFunctionsTest*, which is used to test *FunctionsLib*.

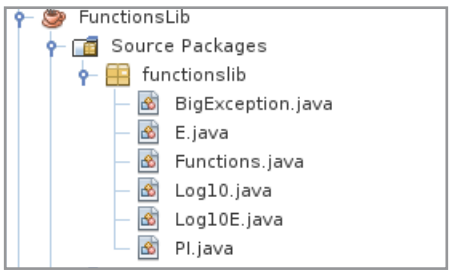
8.3 PROGRAMMING

The following is a description of how the programming has been done, what important decisions have taken and the changes in relation to the above.

A real number is represented as a *BigDecimal*, which offers the basic arithmetic, and when the machine is used, it must be set to work with a certain number of significant digits. The task formulation states that the machine should be able to work with any number of significant digits, but in the current implementation it is limited to 1000000, but in practice it far exceeds what can be used for something, primarily due to performance, but also because it is difficult to present such large numbers in the user interface.

FunctionsLib

I want to start with the *FunctionsLib* class library, which contains 6 classes:



The first *BigException* is trivial and defines an exception type. The class *Functions* is the class with the algorithms, where the other four define mathematical constants:

1. *E* represents the constant with up to 1000000 decimals
2. *PI* represents the constant with up to 1000000 decimals
3. *Log10* represents the natural logaritm of 10 with up to 1000000 decimals
4. *Log10E* represents the 10-number logaritm of with up to 1000000 decimals

These constants should be taken from somewhere, and they can be downloaded from, for example

`http://www.numberworld.org/constants.html`

Here you can not find the last one, but $log_{10}(x) = \frac{log(x)}{log(10)}$, where is the natural logarithm. That is why $log_{10}(e) = \frac{1}{log(10)}$ and can therefore be determined by the third of the above

constants. Below I have shown a method that I used to create strings for the first three constants based on the data I downloaded:

```
private static void createConst(String input, String output)
{
 try
 {
  BufferedReader reader = new BufferedReader(new FileReader(input));
  BufferedWriter writer = new BufferedWriter(new FileWriter(output));
  char[] buff = new char[1000];
  int n = 0;
  StringBuilder builder = new StringBuilder();
  while (builder.length() < 1000000)
  {
   n = reader.read(buff);
   for (int i = 0; i < n; ++i)
    if (buff[i] >= '0' && buff[i] <= '9') builder.append(buff[i]);
  }
  builder.deleteCharAt(0);
  n = 0;
  for (int i = 0; i < builder.length(); i += 1000)
  {
   String line = builder.substring(i, Math.min(i + 1000, builder.length()));
   if (line.length() == 1000)
   {
    writer.write("\"" + line + "\",\n");
    n += line.length();
   }
  }
  writer.close();
 }
 catch (Exception ex)
 {
  System.out.println(ex);
 }
}
```

The method is simple and its parameters indicate the name of the file containing the data (the file downloaded), while the other contains the result, there are 1000 Java strings as lines of 1000 digits. The following method creates the digets for the number $log_{10}(e)$:

```
private static void createLog10E(String output)
{
 try
 {
  BigDecimal x = Log10.value(1000000);
  BigDecimal y =
```

```
    BigDecimal.ONE.divide(x, new MathContext(1000000, RoundingMode.HALF_EVEN));
   BufferedWriter writer = new BufferedWriter(new FileWriter(output));
   writer.write(y.toString());
   writer.close();
  }
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
}
```

You should note that this method can not be performed before the class *Log10* is implemented. The parameter is the name of the file with the result, and with this file available, you can create the text lines to the last constant using the method *createConst()*.

As an example, below is shown some of the code for the class *PI*:

```
package functionslib;

import java.math.*;

public class PI
{
 public static BigDecimal value(int n) throws BigException
 {
  if (n < 1 || n > 1000000) throw new BigException("Can not create PI");
  StringBuilder builder = new StringBuilder("3.");
  ++n;
  for (int i = 0; builder.length() <= n && i < digets.length; ++i)
   builder.append(digets[i]);
  return new BigDecimal(builder.substring(0, n));
 }

 private static final String[] digets = {
  "14159265358979323846264338327950288419 …. ",
  …
 };
}
```

The class has only one method that returns *PI* as a *BigDecimal* with a certain number of decimal places. The three other classes to constants are in principle identical.

Then there is the class *Functions*, where the code fills a lot and I do not want to display the code here. In principle, it is just about implementing the formulas from the design, but as mentioned, some of the formulas result in a poor performance when the argument is given asymptotic values. In order to achieve a reasonable result, it may therefore be necessary to treat special cases for certain values of x. Finally, most of the methods can raise an exception.

The class has a method for each function, and as an example, I will show the code for the function *exp()*. According to the design, the value of the exponential function in an argument x can be determined from the formula:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

The formula is generally effective, as the factorial grows much faster than x elevated in a power. At least as long x is not too big (or goes against minus infinity). If that is the case, $n!$ will at a time be greater than x^n, but it may take a long time, and therefore the fraction only converts slowly to 0. To solve this problem, you can for large values of x use that

$$x = (x - n) + n$$

where $n = [x]$ is the integer part of x, and $|x - n| < 1$. Then is $exp(x) = exp(x - n) \cdot e^n$. One can then take advantage of the above series of reasonably fast converging if $|x| < 1$ and in the case where n can be converted to an *int*, the class *BigDecimal* has a method that can calculate e^n. It probably limits the possibilities for the size of *x*, but if *x* has a size where *n* can not be represented by an *int*, the calculations actually do not give the meaning as it will result in an overflow (exponential function grows very fast). Similar considerations and rewrites can be made in the case where the argument is negative, and corresponding to these remarks, the function *exp()* can be implemented as follows:

```
public BigDecimal exp(BigDecimal x) throws BigException
{
 if (x.abs().compareTo(MAX) >= 0)
  throw new BigException("Illegal argument to exp");
 BigDecimal p = floor(x.abs());
 int m = (x.signum() < 0 ? p.negate() : p).intValue();
 if (m != 0)
 {
  if (x.signum() < 0) x = x.add(p, mc); else x = x.subtract(p, mc);
 }
 BigDecimal y = x.add(BigDecimal.ONE);
 BigDecimal n = BigDecimal.ONE;
 BigDecimal f = BigDecimal.ONE;
 BigDecimal z = x;
 while (true)
 {
  z = z.multiply(x, mc);
  n = n.add(BigDecimal.ONE);
  f = f.multiply(n, mc);
  BigDecimal e = z.divide(f, mc);
  if (e.abs().compareTo(epsilon) <= 0) break;
  y = y.add(e, mc);
 }
 if (m > 0) y = y.multiply(E.value(precision).pow(m, mc), mc);
 else if (m < 0) y = y.divide(E.value(precision).pow(Math.abs(m), mc), mc);
 return scale(y);
}
```

In short, the following happens. First, the value of *x* is tested. If the numerical is too large, an exception is raised because the calculation does not make sense. The following lines should take into account the above transformations of *x* in the case where the numerical value is greater than 1:

```
BigDecimal p = floor(x.abs());
int m = (x.signum() < 0 ? p.negate() : p).intValue();
if (m != 0)
{
```

```
 if (x.signum() < 0) x = x.add(p, mc); else x = x.subtract(p, mc);
}
```

Here you should note the function *floor()*, which is another function in the class *Functions*. The next 13 lines implement the row development of *exp(x)*, which continues until one element in the row becomes negligible. The two next lines again must multiply the result with while the last statement rounds to the desired number of decimal places.

I do not want to show the code for the other methods, but the class is defined as follows:

```
public class Functions
{
 private static final java.util.Random rand = new java.util.Random();
 private static final BigDecimal TWO = new BigDecimal("2");
 private static final BigDecimal HALF = new BigDecimal("0.5");
 private static final BigDecimal MAX = new BigDecimal("999999999");
 private int precision;
 private MathContext mc;
 private final BigDecimal epsilon;
```

```
public Functions(int n) throws BigException
{
 precision = n;
 mc = new MathContext(n + 2);
 epsilon = new BigDecimal(BigInteger.ONE, n);
}
```

Initially, some constants are defined, which are often used in the implementation of the algorithms, while the variable *precision* indicates the number of significant digits. *mc* is an object of the type *MathContext*, which is used by the individual calculations to specify the precision and rounding method, and finally indicates *epsilon* an approximated value of 0. You are encouraged to study the implementation of the other methods. Most are trivial (see the design) as they use already implemented functions or simply simple calculations, while others in principle look similar to the above.

The project BigFunctionsTest

This project creates a console application and is used to test the above library. There are two things to be tested, namely, if the functions returns the correct values, and the performance. To this end, the project has a number of test classes, and as an example, I will show the class *LogTest*, which tests the functions *log()* and *exp()* – where I have only shown parts of the code:

```
ackage bigfunctionstest;

mport java.math.*;
mport functionslib.*;

ublic class LogTest extends Test

public LogTest()
{
 super(1000);
 test1();
 test2(F().random(100), 100);
 test3(F().random(4), 100);
}

private void test1()
{
 try
 {
  System.out.println(F().exp(new BigDecimal("0")).toPlainString());
```

```
   System.out.println(F().exp(new BigDecimal("1")).toPlainString());
   System.out.println(F().exp(new BigDecimal("0.25")));
   System.out.println(F().exp(new BigDecimal("1.5")));
   …
   System.out.println(F().exp(new BigDecimal("-12345.6789")));
   System.out.println("-----------------------------------------");
   System.out.println(F().log(E.value(10000)));
   System.out.println(F().log(new BigDecimal("1")));
   System.out.println(F().log(new BigDecimal("10000000000000000")));
   System.out.println(F().log(new BigDecimal(
   "12345678901234567890123456789012345678.901
234567890123456789012345678901234567890")));
   …
   System.out.println("-----------------------------------------");
   System.out.println(E.value(10000) + ":");
   System.out.println(F().exp(F().log(E.value(10000))));
   System.out.println(new BigDecimal("1") + ":");
   System.out.println(F().exp(F().log(new BigDecimal("1"))));
   System.out.println(new BigDecimal("10000000000000000") + ":");
   System.out.println(F().exp(F().log(new BigDecimal("10000000000000000"))));
   …
   System.out.println("-----------------------------------------");
   System.out.println(F().log10(new BigDecimal("0")));
  }
  catch (Exception ex)
  {
   System.out.println(ex);
   System.out.println("-----------------------------------------");
  }
 }

 private void test2(BigDecimal value, int n)
 {
  try
  {
   long t1 = getTime();
   for (int i = 0; i < n; ++i) F().log(value);
   long t2 = getTime();
   System.out.println(String.format(
    "Performance: %d iteratations, %d milliseconds", n, t2 - t1));
   System.out.println(value);
  }
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
 }
```

```
 private void test3(BigDecimal value, int n)
 {
  try
  {
   long t1 = getTime();
   for (int i = 0; i < n; ++i) F().exp(value);
   long t2 = getTime();
   System.out.println(String.format(
    "Performance: %d iteratations, %d milliseconds", n, t2 - t1));
   System.out.println(value);
  }
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
 }
}
```

First, note that the class inherits a class *Test*. It's a simple class that does nothing but create a *Functions* object and defines a method that can read the machine clock. The class *LogTest* has three test methods, all of which are called from the constructor. The first method tests whether the functions *log()* and *exp()* returns the correct value, and finally the composition

of the two functions is tested, where the result should be the same value as the functions are applied to. The other two test methods are used to test the performance of the logarithm function and exponential function, respectively. In this case, numbers with 1000 digits are created and the function *test2()* determines the logarithm of a random number with 100 digits in front of the decimal point. This operation is performed 100 times. Similarly, *test3()* performs the exponential function on a random number with 3 digits in front of the decimal point, and this operation is also performed 100 times. Done on my machine, the first test has taken approximate 13 seconds while the second test has taken just 2 seconds. The result for the exponential function is fine, but the implementation of the logarithm is the weak link, and it may be a proper task to seek a better implementation of this function. The same goes for *atan()*, where you can also try to find a better algorithm.

The program has other test classes for testing the other functions, and the classes work in principle in the same way as *LogTest*.

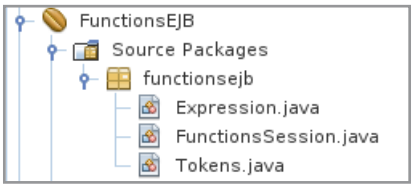
The project FunctionsRemote

This project is again a class library that defines a remote interface for the EJB to write. The library has only a single interface, which defines two methods:

```
package functionsejb;

import java.util.List;
import java.math.BigDecimal;
import javax.ejb.Remote;

@Remote
public interface FunctionsSessionRemote
{
 BigDecimal evaluate(String text, int precision) throws Exception;
 List<String> reserved();
}
```

The project FunctionsEJB

This project is an EJB module project and has a single EJB, which is a stateless session bean. The project consists of three files, with the middle being the bean class, while the other two are the classes for the treatment of an expression:



You should note that the project should have a reference to the jar file from the *FunctionsLib* project.

In relation to the analysis and the design, the project has been expanded to support an additional 9 functions:

- *random()*, which is a random generator that returns a random number between 0 and 1. The method is overwritten with a parameter, that is the number of digits in front of the decimal point.
- *sinh(), asinh(), cosh(), acosh(), tanh(), atanh(), coth()* and *acoth()*, which are the 8 hyperbolic functons.

The classes *Tokens* (which is a file with many classes) and *Expression* can be copied from the *Calc* project in the book Java 3. They need to be changed a bit, and, for example, all variables, parameters and return values of the type *double* must be changed to *BigDecimal*. In addition, new token classes must be defined for the new functions, and the *VarToken* class should be deleted. In the *Expression* class, there are only modest changes, and you are encouraged to investigate what has been changed.

Then there is the *FunctionSession* class that implements the above remote interface:

```
package functionsejb;

import java.util.List;
import java.math.BigDecimal;
import javax.ejb.Stateless;

@Stateless
public class FunctionsSession implements FunctionsSessionRemote
{
 public BigDecimal evaluate(String text, int precision) throws Exception
 {
  Expression expression = new Expression(text, precision);
  return expression.getValue();
 }
```

```
 public List<String> reserved()
 {
  return Tokens.getNames();
 }
}
```

The class is trivial as all the work is in the above two classes – and of course, the class *Functions* in the *FunctionsLib* library.

Then the EJB module is ready and can be deployed to the Glassfish server.

BigCalc

Finally, there is the web application. The user interface is essentially the same as shown during the analysis, and there is only one extension, since the values stored with the *STO* button appear in a table below the page's buttons. The expression that the program has to calculate is entered in the lower entry field, while the top shows the result of the calculation. To save values in variables (*STO*), you must enter a name in the field after *STO*. The name must start with a letter, followed by up to 4 characters, which are letters and numbers. There is

no separation between upper and lowercase letters and the name must not be a reserved word (name of a function). The value stored is the value of the result field. If you enter the name of an existing variable and click *RCL*, the value appears in the result field. For example, if I want to save the square root of 2 in a variable *X2*, I have to enter the formula

```
sqrt(2)
```

and click *Calculate*, after which the value is added to the result field. To save the value, you must enter the name *X2* after *STO* and click *STO*. If you now want to apply the variable in a formula, enter the variable as follows:

```
sqr(@x2) + 3
```

When I have chosen this syntax, due to the fact that values of variables can be very long, and inserted directly into a formula, it can easily lead to an expression that is impossible to read.

The application has only one page, which is quite simple, but it has a backing bean that uses my session bean. There is not much new, and I do not want to show the code here, but when you study the code, note that it is this bean that substitutes the value of variables before the expression is sent to my session bean.